Program Office Guide to Ada, Edition 2

C.N. AUSNIT
E.R. ANSAROV
N.H. COHEN
M.V. ZIEMBA, 1 Lt, USAF

SofTech, Inc.
460 Totten Pond Road
Waltham, MA 02254

7 October 1986

Prepared For

ELECTRONIC SYSTEMS DIVISION
AIR FORCE SYSTEMS COMMAND
DEPUTY FOR DEVELOPMENT PLANS
HANSCOM AIR FORCE BASE, MASSACHUSETTS 01731-5000

LEGAL NOTICE

When U. S. Government drawings, specifications or other data are used for any purpose other than a definitely related government procurement operation, the government thereby incurs no responsibility nor any obligation whatsoever; and the fact that the government may have formulated, furnished, or in any way supplied the said drawings, specifications, or other data is not to be regarded by implication or otherwise as in any manner licensing the holder or any other person or conveying any rights or permission to manufacture, use, or sell any patented invention that may in any way be related thereto.

OTHER NOTICES

Do not return this copy. Retain or destroy.

"This technical report has been reviewed and is approved for publication."

MARK V. ZIEMBA, 1Lt, USAF
Project Officer, Software
Engineering Tools & Methods

ARTHUR G. DECELLES, Major, USAF
Program Manager, Computer Resource
Management Technology (PE 64740F)

FOR THE COMMANDER

ROBERT J. KENT
Director
Software Design Center
Deputy for Development Plans
and Support Systems

REPORT DOCUMENTATION PAGE

1a. REPORT SECURITY CLASSIFICATION Unclassified	1b. RESTRICTIVE MARKINGS
2a. SECURITY CLASSIFICATION AUTHORITY	3. DISTRIBUTION / AVAILABILITY OF REPORT Approved For Public Release; Distribution Unlimited
2b. DECLASSIFICATION / DOWNGRADING SCHEDULE	
4. PERFORMING ORGANIZATION REPORT NUMBER(S) 3285-4-253/1.1	5. MONITORING ORGANIZATION REPORT NUMBER(S) ESD-TR-86-282 (II)
6a. NAME OF PERFORMING ORGANIZATION SofTech, Inc. 6b. OFFICE SYMBOL (*If applicable*)	7a. NAME OF MONITORING ORGANIZATION HQ, Electronic Systems Division (XRSE)
6c. ADDRESS (*City, State, and ZIP Code*) 460 Totten Pond Road Waltham, Ma. 02254	7b. ADDRESS (*City, State, and ZIP Code*) Hanscom AFB Massachusetts, 01731
8a. NAME OF FUNDING / SPONSORING ORGANIZATION Deputy for Development Plans 8b. OFFICE SYMBOL (*If applicable*) ESD/XRSE	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER F33600-84-D-0280
8c. ADDRESS (*City, State, and ZIP Code*) Hanscom AFB Massachusetts, 01731	10. SOURCE OF FUNDING NUMBERS PROGRAM ELEMENT NO. / PROJECT NO. / TASK NO. / WORK UNIT ACCESSION NO.
11. TITLE (*Include Security Classification*) Program Office Guide to Ada, Edition 2	
12. PERSONAL AUTHOR(S) C.N. Ausnit, E.R. Ansarov, N.H. Cohen, M.V. Ziemba	
13a. TYPE OF REPORT Technical 13b. TIME COVERED FROM _____ TO _____	14. DATE OF REPORT (*Year, Month, Day*) 1986 October 7 15. PAGE COUNT 86
16. SUPPLEMENTARY NOTATION	

17. COSATI CODES			18. SUBJECT TERMS (*Continue on reverse if necessary and identify by block number*)
FIELD	GROUP	SUB-GROUP	Ada, Ada Compiler, AJPO, Run-Time Support, Computer Languages

19. ABSTRACT (*Continue on reverse if necessary and identify by block number*)

The purpose of the Program Office Guide to Ada is to discuss issues affecting the selection, development, and maintenance of systems whose software is written in the Ada language. Each volume focuses on a different set of topics and their implications for managers. This edition concentrates on: MIL-STD-2167, MIL-STD-2168, proposal evaluation, reusability, portability, estimating and development efforts, benchmarks, and software libraries.

20. DISTRIBUTION / AVAILABILITY OF ABSTRACT ☐ UNCLASSIFIED/UNLIMITED ☒ SAME AS RPT. ☐ DTIC USERS	21. ABSTRACT SECURITY CLASSIFICATION Unclassified
22a. NAME OF RESPONSIBLE INDIVIDUAL M.V. Ziemba, 1Lt, USAF	22b. TELEPHONE (*Include Area Code*) (617) 377-2656 22c. OFFICE SYMBOL ESD/XRSE

ACKNOWLEDGEMENT

This report was prepared by SofTech, Inc. The work reported here was sponsored by the Directorate of Software Engineering, Deputy for Development Plans and Support Systems of the Electronic Systems Division (ESD) of the United States Air Force Systems Command, Hanscom Air Force Base, Ma 01731. Continuing funding for this work was provided by the Air Force Computer Resource Management Technology Program, Program Element 64740F.

The Computer Resource Management Technology Program supports the development and transition into active use of tools and techniques needed to cope with the explosive growth in Air Force systems that use computer resources. The objectives of this Program are:

to provide for the transition to Air Force systems of computer system developments in laboratories, industry, and academia;

to develop and apply software acquisition management techniques to reduce life cycle costs;

to provide improved software design tools;

to address problems associated with computer security;

to develop advanced software engineering tools, techniques, and systems;

to support the implementation of high-order programming languages;

to improve human engineering of computer systems; and

to develop and apply computer simulation techniques in support of systems acquisition.

TRADEMARKS

Ada is a registered trademark of the U.S. Government (Ada Joint Program Office)

AdaGRAPH is a registered trademark of The Analytic Sciences Corporation.

Byron is a registered trademark of Intermetrics, Inc.

GRACE is a trademark of EVB Software Engineering, Inc.

Math Advantage is a registered trademark of Quantitative Technology Corporation.

PAMELA is a trademark of George W. Cherry.

EXECUTIVE SUMMARY

The successful development of complex software requires application of a software engineering methodology. DoD-STD-2167, a new standard for software development and documentation, can be used in conjunction with such a methodology. The standard addresses activities and products associated with each stage of the traditional "waterfall" life cycle. DoD-STD-2167 was written with large, complex systems in mind, but it can be tailored for use on smaller projects by deleting certain requirements. In selecting a methodology, one should consider its technical characteristics, management characteristics, compatibility with Ada, and compatibility with the development organization. Methodologies may be supported by metnods and tools. Methods and tools well suited to Ada include PAMELA, SHARP, Byron, DARTS II, and OOD.

A new standard for software quality assurance, DoD-STD-2168, can be used in conjunction with DoD-STD-2167. DoD-STD-2168 describes evaluation of the activities and products described in DoD-STD-2167, as well as the administration of quality-assurance measures. It is a general framework, allowing details to be filled in differently for different projects. Like DoD-STD-2167, DoD-STD-2168 can be tailored by deletion of certain requirements.

Special considerations come into play when evaluating proposals for Ada software developments. It must be ascertained whether the offerer understands the appropriate use of Ada, risks associated with the use of Ada, and Air Force policy regarding Ada. Ada training may be necessary to ensure that both technical and management positions are staffed by people with appropriate qualifications. Corporate experience with the Ada language may be as important as corporate experience with the application. The choice of a programming environment (including an Ada compiler) may affect the success of a project. Schedule predictions require careful justification because of the lack of extensive prior experience with Ada development.

Ada can reduce software costs by facilitating reusability and portability. Reusable software is typically more expensive to produce and less efficient than single-use software, so managers must decide both whether to use reusable software and whether to produce reusable software. Reusability can be enhanced by appropriate design and coding techniques. The risks associated with reusable software can be reduced by providing incentives for the development of reusable-software libraries by projects, contractors, off-the-shelf software vendors, and voluntary contributors. Portability is a relative property

rather than an absolute one, for three reasons: a program may behave in acceptably equivalent ways in two environments even if it does not behave identically in each case; a program may consist of both portable and nonportable modules; and a program might port only to a class of target environments sharing certain commonplace properties. Portability can be enhanced by appropriate requirements definition, design, and coding.

Several models, including COCOMO, SLIM, Jensen's model, and PRICE-S, have been used to predict software development costs. The calculations based on these models include numbers based on the programming language and the programming environment, among other factors. There is not yet enough empirical data about Ada software development to ascertain whether the models can accurately predict Ada software development costs, or what the appropriate numbers are for the Ada language and Ada programming environments. There is a consensus that use of Ada will increase software development costs in the short run, then decrease them as experience with the language grows.

The Ada Joint Program Office Evaluation and Validation (E&V) task is charged with providing a basis for evaluating Ada Programming Support Environments. This entails both evaluation of an environment as a whole and evaluation or validation of individual environment components. The most prominent component in an Ada Programming Support Environment is an Ada compiler. Just as the Ada Compiler Validation capability validates a compiler's conformance to the Ada standard, a prototype Ada Compiler Evaluation Capability provides benchmarks to evaluate the performance of a compiler's object code.

Reuse of Ada software requires the development of software libraries. Both general-purpose and special-purpose libraries of reusable Ada parts have been developed commercially. Another library, the Ada Repository, is maintained in the public domain by the Department of Defense.

Table of Contents

List of Figures

SECTION 8

MIL-STD-2167 AND METHODOLOGIES FOR USE WITH ADA

The complexity of today's software systems requires the use of good development methods and tools. Furthermore, the development process must be well documented to facilitate operation and maintenance of the system. Software-engineering methodologies have existed since the 1960's; however, with the realization of the software crisis and the advent of Ada, there is now a much sharper focus on software development as an engineering discipline and on the tools needed for this discipline.

Not all methods are suited to all applications. This section presents the attributes by which methodologies can be evaluated (Section 8.1) and discusses these characteristics in relation to DoD-STD-2167, the new documentation standard governing military software development. Certain methodologies and tools especially appropriate for use with Ada are briefly described.

8.1 Overview

Today's software systems are characterized by a high level of complexity. Often these systems are targeted to previously unknown application areas, which introduces uncertainty about the system's intended functionality and performance. Thus development of these systems is a high-risk activity. To cope with complexity, uncertainty, and the risks involved, system developers use various techniques, methods, and methodologies.

Three crucial activities in building software systems are modeling, information accumulation, and analysis. These activities require the use of software methods and methodologies. A software method is a collection of rules, guidelines, and techniques that define a disciplined process for producing software. A software methodology is collection of such methods. It provides general principles, practices and procedures for using software and management technologies to produce high-quality software on schedule with minimal cost.

Software methodologies fall into four classes:

- Product-oriented methodologies

- Data-structuring methodologies
- Prototyping methodologies
- Object-oriented methodologies

Most of the currently used methodologies are product-oriented. These methodologies emphasize the products obtained at the different stages of system development, rather than the activities used to prepare and analyze these products. The product-oriented methodologies tend to divide the life cycle of the software into well-specified phases. The products of one phase must be finished and reviewed before the next phase can begin. In this traditional view of the software life cycle (sometimes called the "waterfall" model), the phases are typically identified as follows:

- Requirements analysis and specification
- Preliminary design
- Detailed design
- Implementation
- Testing and integration
- Maintenance

The product-oriented methodologies have been successfully applied since the mid-1960's. They are management-oriented, because the definition of the products and their review points can be seen as milestones that provide management information and control. DoD-STD-2167 (described in Section 8.2) recommends applying a product-oriented methodology for the development of defense software.

The product-oriented methodologies are not always the most effective. For example, if the application area is unfamiliar, then the transition from high-level design to low-level implementation is difficult. This is especially true when a new technology is introduced into a product-oriented development approach. The other classes of methodologies were developed in response to these problems.

Data-structuring methodologies emphasize the structure of the data being processed and the composition of the processing steps that perform the intended data transformations. These methodologies are best used for applications involving complex data or file structures.

Prototyping methodologies have traditionally been used to support product-oriented methodologies in clarifying the software requirements. The approach is to build an incomplete version of

the system initially and then to provide the full, required functionality gradually through refinement.

Object-oriented methodologies first identify the abstract data objects to be manipulated by the program and the operations that are performed on these objects. The organization and operation of the software is then derived from a model describing the interactions between the objects. This approach can be very suitable for smaller projects, but its effectiveness decreases as system size and complexity increase. Object-oriented methodologies are well supported by the package and private-type features of the Ada language. There are currently few actual object-oriented methodologies to guide the developer. One widely publicized object-oriented method is sketched in Section 8.4.4.

8.2 Characteristics of Methods: MIL-STD-2167 View

DoD-STD-2167 defines standard approaches to be used in the development and documentation of defense software in mission-critical systems, but it is also applicable to other data processing systems. Standard development and documentation procedures are presented in the framework of a system life cycle and a software development cycle. DoD-STD-2167 recognizes the phases of the traditional life-cycle.

The two main sections in the body of DoD-STD-2167 are the "General Requirements" (Section 4), where the software life cycle is explained on a higher level, and "Detailed Requirements" (Section 5). The section "Detailed Requirements" identifies the activities, products, evaluations, and baselines of each phase of the life cycle. Other topics addressed in the "Detailed Requirements" section are software configuration, software quality, and software project management. A description of the system life cycle, design and coding standards, and guidelines for tailoring the standards in this document appear in Appendices B, C, and D respectively. A set of Data Item Descriptions (DID's) gives detailed documentation requirements.

Associated with DoD-STD-2167 is the military handbook, DoD-HDBK-287. The latest draft of the military handbook was prepared in May 1986. DoD-HDBK-287 supersedes DoD-HDBK-281 since October 1984. This military handbook is designed to provide guidance in applying DoD-STD-2167, MIL-STD-483, MIL-STD-490, and MIL-STD-1521 to software acquisition, development, and support. The handbook emphasizes the tailoring process.

DoD-STD-2167 supersedes the following documents:

- MIL-STD-490
- MIL-STD-483
- DoD-STD-1679 (Navy)

- MIL-STD-1644B (TD)

The latest revision of DoD-STD-2167 was released in June 1985. There have been disagreements on some Ada-related aspects of this revision, and major changes are likely to follow. Revision A will be released for formal government and industry review in September 1986. It is expected to be approved and released in final form in the fall of 1987.

The Air Force has issued a memorandum providing an interim policy and guidance on implementing DoD-STD-2167. A package containing guidelines for applying the standard is available from:

Roland Usher
U.S. Air Force
Electronic Systems Division
ESD/PLEA
Hanscom AFB, MA

(617) 377-4002

The package includes:

- Areas of concern
- A requirements application matrix
- A-Spec mapping from MIL-STD-490 to MIL-STD-490A
- Sample SOW's
- Sample CRDL's
- An RFP checklist
- CRWG policies
- Schedules

8.2.1 DoD-STD-2167 View of Analysis

The goal of requirements analysis, as set forth in DoD-STD-2167, is to establish a complete set of functional, performance, interface, and qualification requirements for each software item. The analysis activities should check that documents are both adequate and complete, and that they can be readily tested and understood. DoD-STD-2167 requires the use of a structured requirements analysis tool or technique as well as the development of specification sets. The requirements analysis should produce:

- Records of analysis activities, including document reviews, studies, and action items
- A software-requirements specification
- An interface-requirements specification

Software specifications are reviewed during requirements analysis.

8.2.2 DoD-STD-2167 View of Design

Software design translates the requirements specifications into a blueprint of the system. This blueprint represents a model of the software that is to be developed. Design activity can be viewed as consisting of two major subphases: architectural, or preliminary, design and detailed design. Architectural design emphasizes the structure of the system -- the decomposition of the system into modules and the interfaces between the modules. The detailed design emphasizes selection and evaluation of the algorithms necessary to perform the processing specified for each module.

The preliminary design activities according to DoD-STD-2167 are:

- Assignment of functional and performance software requirements to top-level software components
- Use of a top-down design approach
- Establishment of a top-level design determining flow of data and flow of control between components and specifying top-level algorithms
- Use of a program-design language (PDL) to express the top-level design
- Development of test plans
- Development of user manuals

The above mentioned activities should produce:

- Records of reviews
- A top-level software design document
- A software test plan
- Various manuals

The preliminary design review should review design products and demonstrate traceability back to the requirements.

The activities that DoD-STD-2167 defines for detailed design are:

- Refinement of top-level software components to lower levels
- Use of a top-down approach
- Use of a bottom-up approach only in critical lower-level units
- Use of a PDL
- Establishment of software development files (SDF's) for all software units
- Development of tests for each unit
- Development of integration-test classes

The detailed design products are:

- A software detailed-design document
- An interface design document
- A database design document
- Software test descriptions
- Software development files (SDF's)
- If required by the contractor, a software programmer's manual or a firmware-support manual

Each of these products is reviewed during the detailed design phase.

8.2.3 Evaluation of DoD-STD-2167

If a standard is too rigid, it will not be very applicable to the varied requirements of most systems. However, if the standard is too vague, it will be hard to enforce. DoD-STD-2167's Appendix D, "Tailoring Guidelines For This Standard," avoids the two extremes by illustrating alternative approaches depending on the kind of software to be developed.

DoD-STD-2167 is intended for use during the life cycle of mission-critical systems, which are typically complex systems requiring complex development procedures. To compensate for this when applying the standard to less complex systems, the tailoring

guidelines given in Appendix D allow for many documents to be combined with others or abridged.

Compared to the previous documentation requirements, the latest revision of DoD-STD-2167 takes a big step towards compatibility with Ada. Nonetheless, certain incompatibilities remain. In particular, the design and coding standards in DoD-STD-2167 Appendix B and the DID for the Software Detailed Design Document are not completely appropriate for use with Ada. Imposing them on an Ada project could result in design constraints that will increase the overall costs.

For example, in Section 4.2 of DoD-STD-2167, software unit design is required to be presented in a top-down manner and to form tree-like structures. There are many methods, such as object-oriented design, prototyping, and reuse of general-purpose software parts, that are compatible with Ada but not necessarily top-down or tree-like in structure. For many application areas these methods might be more effective than the traditional top-down ones. The section on the preliminary design methods (5.2) requires that a program design language (PDL) be used in developing the top-level design. This rule might be limiting in light of current practice, which is to use a set of methods to bridge the gap between requirements analysis and detailed design, eventually resulting in a PDL representation. These methods produce three kinds of products: graphical representations, data dictionaries, and textual representations. Only textual representations are most appropriately expressed in a PDL.

8.3 Selection Guidelines

The criteria for selecting a methodology should be based on the specific needs of an organization and project. Methodologies can be described and classified according to four kinds of characteristics:

- Technical characteristics affect how well a methodology supports the development of a product meeting desirable goals such as reliability, adaptability, understandability, reusability, and portability. Such characteristics might include the use of formal proof techniques, the definition of a real-world model to provide the basis for defining software functions, design guidelines, or coding guidelines.

- Management characteristics affect the support that a methodology provides for planning, organizing and controlling software projects. Examples of management characteristics are the historical accuracy of cost estimation methods, the use of guidelines for division of labor, and the use of cost-accounting mechanisms.

- Ada-compatibility characteristics affect how well a methodology supports and encourages appropriate use of Ada

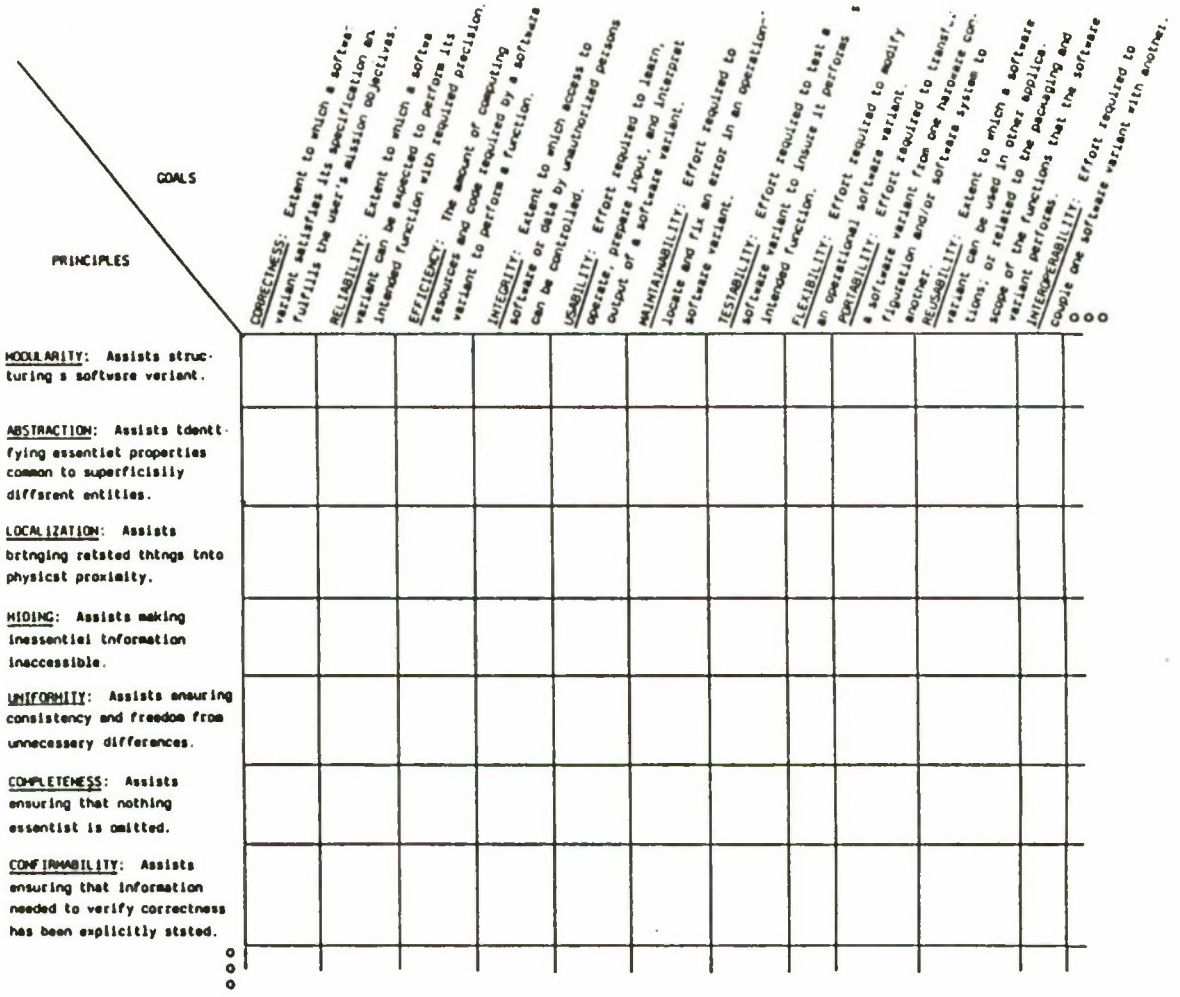

Figure 8-1. Framework for a matrix of technical characteristics of software methodologies. (From [MRY86], page 4-6.)

language concepts and constructs. Examples of Ada-compatibility characteristics are the number of concurrent threads of control typically identified in a design and the similarity of structure between a design and the resulting Ada program.

- Usage characteristics affect how well a methodology suits a particular organization. They cover issues such as availability, purchase, training, and customization.

Given any one of these four categories, characteristics that belong to that category can be organized in a two-dimensional matrix. One dimension represents desirable goals and the other represents software-engineering principles that support those goals. Entries in the matrix are characteristics of software methodologies. A characteristic is listed in the box corresponding to a given principle and goal if the characteristic helps a methodology to apply that principle in support of that goal. Figure 8-1 shows the structure of such a matrix for technical characteristics and Figure 8-2 shows the structure of such a matrix for Ada-compatibility characteristics.

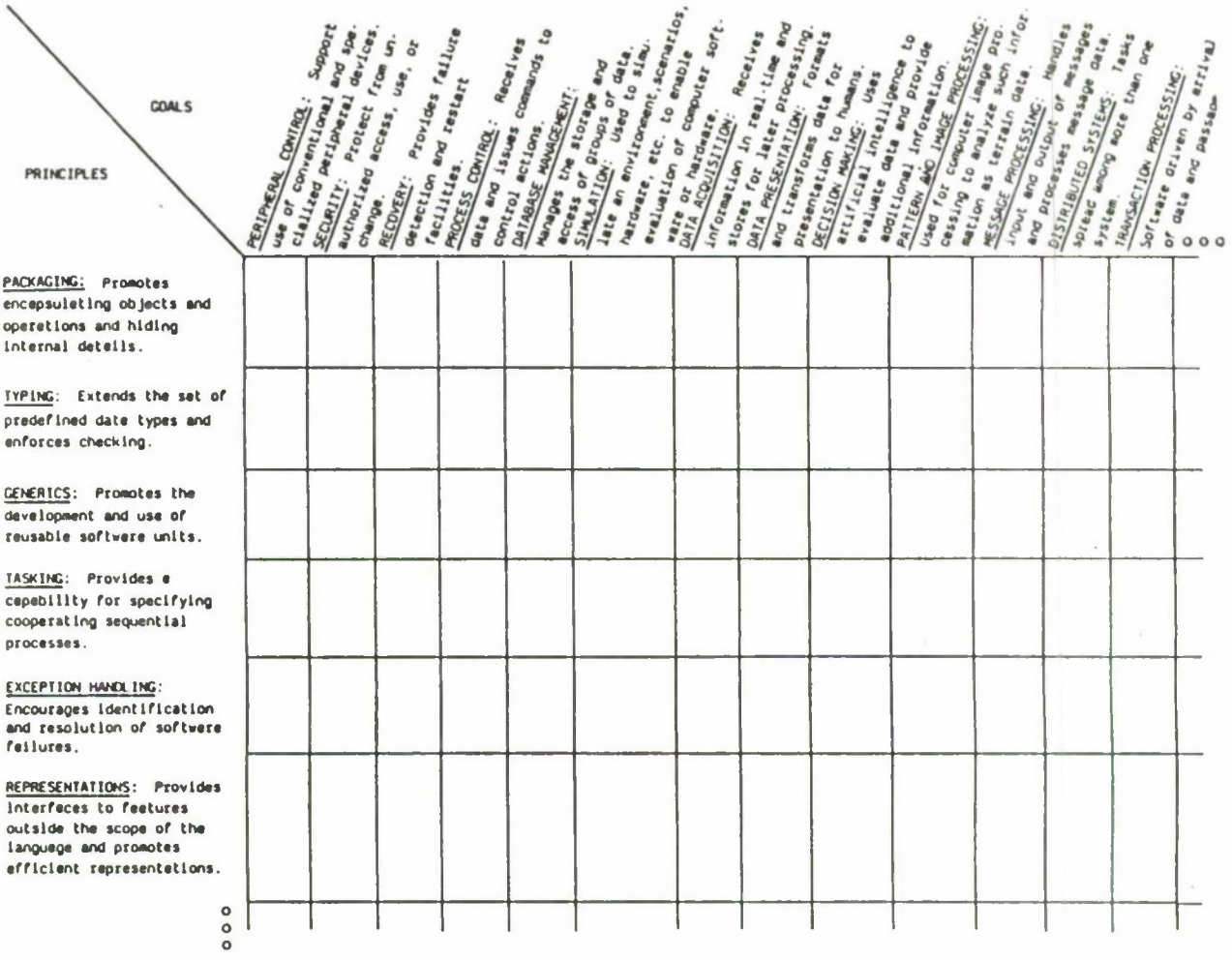

Figure 8-2. Framework for a matrix of Ada-compatibility characteristics of software methodologies. (From [MRY86], page 4-14.)

A more detailed analysis of the subject of a methodology characteristics framework and methodology selection can be found in the STARS Methodology Area Summary [MRY86] prepared for the Office of the Undersecretary of Defense for Research and Engineering by the Institute for Defense Analyses.

8.4 Prominent Ada-Oriented Methods and Tools

The diversity of existing methodologies is enormous. These methodologies are often incompatible with each other and use different tool sets. Tools should be selected based on their support for the chosen methodology. Ideally, the tool set should be integrated to provide a continuous flow of information from one tool to the next, covering most or all of the life cycle.

This section briefly sketches some prominent methods and tools that are especially appropriate for use with Ada. In most cases, the methods and tools were developed in response to the Ada language. The notable exception is Object-Oriented Design

(OOD), which (known by a variety of different names) strongly influenced the design of the Ada language.

8.4.1 PAMELA

Some Ada-specific methods are useful in particular application areas. Process Abstraction Method for Embedded Large Applications (PAMELA) is a process-oriented method especially effective for real-time, concurrent, or parallel applications. PAMELA features a graphical specification of design and a systematic means for transforming the graphical specification and its elements into Ada program units and their elements. This method can be applied across the entire software-development life cycle. There is an interactive graphical tool, called AdaGRAPH, which is used in conjunction with PAMELA. Some features of AdaGRAPH are:

- Construction of a system design expressed in terms of Hierarchical Process Graphs that are guaranteed to be consistent and complete
- One-to-one correspondence between graphical design elements and Ada program elements
- Automatic generation of a software database, module map, and Ada PDL.

More information on PAMELA can be obtained from:

George W. Cherry
P.O. Box 2429
Reston, VA 22090

(703) 427-4450

More information on AdaGRAPH can be obtained from:

The Analytic Sciences Corporation
1700 North Moore Street, Suite 1220
Arlington, VA 22209

(703) 558-7400

8.4.2 SHARP

Structured Hierarchical Ada Representation using Pictographs (SHARP) is another Ada-specific method. It uses pictographs to characterize structured Ada designs. SHARP is an attempt to solve some of the problems associated with using Ada PDL to represent software designs. The purported problems are:

- o System engineers and product managers are usually not familiar with Ada and therefore have difficulty understanding Ada PDL's.

- o It is very costly to use PDL to produce design alternatives during early stages of the design phase.

- o Using Ada PDL forces one into premature coding.

SHARP is based on the pictographs developed by R.J.A. Buhr of Carlton University, Ottawa, Canada. The benefits ascribed to SHARP are:

- o Improved readability of design information within C5 product specifications

- o Rapid development of diagrams presenting alternative Ada designs

- o Assistance in software quality testing by clear indication of test drivers, special test triggers and test recorders, and Ada program units to be tested

- o Ada training

- o Cost/schedule estimates for Ada work

Work on SHARP is being performed under contract to the Air Force. For more information, contact:

Lt. Mark V. Ziemba
ESD/PLSE
Hanscom AFB, MA

AUTOVON 478-2656
MILNET Ziemba.SDruid@RADC-MULTICS.ARPA

8.4.3 Byron and DARTS II

Byron is a software development tool that consists of two parts, the Byron Analyzer and the Byron Document Generator. The Analyzer checks source code for errors and then stores the result in a program library. This library is used by the Byron Document Generator, which performs three functions:

1. It is a text processor producing quality documents.

2. It can collect information that is widely dispersed over many files into a single concise document.

3. It can be used to derive and print information that is not apparent from reading the source code. For example it can list all subprograms calling a program unit.

Design Aids for Real-Time Systems II (DARTS II) is a tool that combines all of the features of Byron with additional capabilities to develop requirements and graphically portray tne hierarchical structure of Ada designs. The advantage of DARTS II is its use of a single specification medium for software that evolves throughout the life cycle. The intention is to reduce development and maintenance costs and to provide a machine-processible mechanism for traceability.

Further information about Byron and DARTS II can be obtained from:

Intermetrics, Inc.
733 Concord Avenue
Cambridge, MA 02138

(617) 661-1840

8.4.4 OOD

Object-Oriented Design (OOD) is the name given by Grady Booch [Boo82] to a method first identified by David Parnas [Par72] and used as the basis for the CLU, Alphard, and Modula-2 programming languages. The method also profoundly influenced the design of Ada. The method is based on identifying abstract data types, each consisting of a set of values and a set of operations applicable to those values. An abstract data type can be implemented by building:

- A data structure to represent the values
- Subprograms to manipulate that data structure in accordance with the operations

The data structure and the operations manipulating it are encapsulated in one region of program text. The remainder of the program manipulates the data type by calling the subprograms, but not exploiting any knowledge of the type's implementation. Much of the work that computer programs do can be described in terms of applying an abstract data type's operations to values in that type.

A program design based on the manipulation of abstract data types can be mapped directly to Ada program units. In the Ada language, the data structure for an abstract data type and the algorithms for manipulating that data structure are naturally encapsulated in a package in which the abstract data type is declared as a private type. An Ada compiler ensures that the implementation of the abstract data type is hidden from other parts of the program.

Grady Booch describes OOD as consisting of the following steps [Boo82]:

1. Define the problem.

2. Develop an informal strategy for the abstract world.

3. Formalize the strategy:

    a. Identify objects and their attributes.

    b. Identify operations on the objects.

    c. Establish the interfaces.

    d. Implement the operations.

Step 1, problem definition, is requirements analysis. Step 2, the development of a strategy for solving the problem, is the most difficult part of the design process, but it is not the focus of OOD. OOD is helpful primarily in mapping an informal strategy to Ada program units once that informal strategy has been devised.

8.5 Implementation Issues

8.5.1 DoD-STD-2167 View

Implementation is the translation of a design into the language of the target computer and into other products needed to create an operational system. The resulting system should meet the resource, accuracy and performance constraints of the specification. To create clear, understandable programs, the implementation phase utilizes techniques such as stepwise decomposition and data abstraction.

The DoD-STD-2167's requirements for implementation activities are:

- Code units in a top-down hierarchical sequence and in accordance with DoD-STD-2167 coding standards.
- Maintain unit development files on all units.
- Test each unit.
- Develop integration and formal testing procedures.

The products of these activities are:

- Source and object code for each unit

- Software test procedures
- Unit development folders
- Software product specifications

There should be reviews of the source code, functional configuration audits, and physical configuration audits during this phase.

8.5.2 Coding Standards and Conventions

Program-wide or project-wide coding standards and conventions can reduce development and maintenance costs. They should be established before the design phase. Standards and conventions should be established both for the program design language (PDL) and the programming language.

DoD-STD-2167 minimum level coding standards are:

- All code shall be written in a higher order language. A waiver is required for the use of assembly language. (Current Air Force policy is more restrictive, specifying Ada as the preferred language, mandating the use of Ada for certain programs, and establishing stringent requirements for the granting of a waiver. See Section 2.1.)
- Only the sequence, if-then-else, case, do-while, and do-until control structures shall be used.
- The source code for a given unit shall not exceed 200 statements and the average length of a unit shall not exceed 100 statements.
- Units shall exhibit the following characteristics:
  - Units shall not share temporary storage locations.
  - A unit shall perform a single function.
  - Code shall not be modified during execution.
  - All unit source code shall include a prologue, declarative statements, then executable statements or comments.
  - Except for error exits, units shall have a single entry and exit.
- Entities shall have meaningful names and symbolic parameters shall be used.
- Mixed mode operations shall be avoided.

The DoD-STD-2167 primary coding standards are not based on Ada, but are language-independent. Some of the standards are impossible to violate in the Ada language. At the same time, these standards do not guarantee correct and consistent usage of the Ada language. Therefore, Revision A of DoD-STD-2167 contains default coding standards to be used for Ada programs unless a project provides its own coding standards.

8.5.3 Tools

A basic set of software development tools consists of a compiler, a linker, a debugger, and, if the target machine is different from the host, an exporter. Other tools that can be also helpful in program development are:

- A language-oriented editor, providing templates that are filled in by the programmer and detecting illegal constructs as they are entered
- Subject to the caveats given in section 7.2, a tool to translate another programming language into Ada
- A formatter, or "pretty printer," which formats Ada source code according to the rules set by the Ada standard and by conventions
- Tools to display the structure of an Ada program graphically, including the nesting structure, subprogram calling patterns, task calling patterns, compilation-order dependencies, and task master/dependent relationships
- Tools to document which exceptions can be raised by which subprograms and entries
- Tools to document which compilation units depend, directly or indirectly, upon others, and thus which units have to be recompiled when a given unit is modified
- A tool to generate cross-reference listings, if this is not done by the compiler
- Tools to monitor program performance and locate bottlenecks
- A tool to determine which parts of Ada source code have not been exercised by test cases
- Tools to maintain test plans and automate the administration of tests
- Tools to maintain and retrieve program documentation
- Configuration-management and library-management tools

8.6 Testing Methods

DoD-STD-2167 approach to system/component testing is very general. The default testing method is top-down. If the vendor proposes alternative methodologies and the contracting agency approves them, the vendor may depart from a top-down testing approach. This situation is very likely to occur when integrating or testing critical units, or incorporating commercially available, reusable, and Government-furnished software. The vendor must describe the criteria, such as performance, cost, and schedule, for determining which units are critical.

DoD-STD-2167 puts a strong emphasis on complete unit-level testing. This approach ensures higher reliability, but may be expensive if it is carried to an extreme. This is particularly true for highly modular, well designed Ada software. It may not be necessary to unit-test all Ada programing units, especially if they contain only data or consist only of short sequences of calls and associated exception handlers. In the IBM "cleanroom" method [Dye83], unit testing is replaced by formal and semiformal mathematical approaches to ensuring correctness, applied throughout specification, design, and coding.

8.7 Summary: Interaction Between Methods and the Life Cycle

Methodologies that incorporate software engineering techniques are necessary for coping with the risk, uncertainty, and complexity associated with large software projects. An ideal methodology would cover the entire life cycle by addressing all concerns associated with each phase. In reality, no current methodology addresses all of these concerns completely.

Figure 8-3 describes some of the prevailing software methodologies, methods, and tools, including the degree to which they address various technical characteristics, the range of the life cycle over which they are applicable, and range of task complexity. The six technical characteristics considered are:

- Function Hierarchy: the description of an activity at one level in terms of several interconnected activities at a greater level of detail
- Data Hierarchy: the description of data at one level in terms of several interrelated pieces of data at a greater level of detail
- Interfaces: distinct and well-defined boundaries between processes or sets of data
- Control Flow: the sequence in which activities occur

LIFE CYCLE		TYPE	TECHNICAL CHARACTERISTICS							
			Function Hierarchy	Data Hierarchy	Interfaces	Control Flow	Data Flow	Data Abstraction	Life-Cycle [1] Range	Task [2] Range
ANALYSIS	PSL/PSA	tool	much	much	some	some	some	some	RD + DD	A → D
	SADT FOR ANALYSIS	methodology	much	much	much	some	much	much	NA → DD	B → E
	STRUCTURED SYSTEMS ANALYSIS	method	some	much	much	none	much	some	NA → AD	B → D
	SREM	methodology	much	some	much	some	some	some	RD → AD	D → E
DESIGN	HIPO	tool	much	none	some	none	much	none	RD → AD	A → C
	JACKSON DESIGN	method	much	much	none	some	some	some	DD → P	A → C
	STRUCTURED DESIGN	method	much	none	much	none	much	some	AD → DD	B → D
	SADT FOR DESIGN	methodology	much	much	much	some	much	much	NA → DD	C → E
	WARNIER/ORR DESIGN	method	much	much	some	some	some	some	AD → P	B → D
	OBJECT-ORIENTED DESIGN	method	some	some	much	none	none	much	AD	A → C
CONSTRUCTION	N-S/CHAPIN CHARTS	tool	some	none	none	much	none	none	DD → P	A → C
	PROGRAM DESIGN LANGUAGE	tool	some	some	some	much	none	none	DD → P	A → D
	STRUCTURED PROGRAMMING	method	much	none	some	much	none	none	P	A → D
	STRUCTURED WALK-THRUs	methodology	N/A	N/A	much	much	much	N/A	RD → T	A → E
	TOP DOWN IMPLEMENTATION & TESTING	method	much	N/A	much	much	some	none	DD → T	B → D

[1] X → Y = phase X thru phase Y; X + Y = phase X and phase Y

[2]

Phases	Abbreviations	Task Complexity
needs analysis	NA	A = very simple
requirements definition	RD	B = simple
architectural design	AD	C = average
detailed design	DD	D = complex
programming	P	E = very complex
testing	T	

Figure 8-3. Evaluation of software-engineering techniques. The six technical characteristics are explained in the text. Life cycle phases are needs analysis (NA), requirements definition (RD), architectural design (AD), detailed design (DD), programming (P), and testing (T). A life-cycle range of the form X->Y means phase X through phase Y, while X+Y means phase X and phase Y. Task complexity is rated on a scale from A (very simple) to E (very complex).

- o Data Flow: the flow of information among various system elements

- o Data Abstraction: describing data manipulation in terms of abstract operations, without regard to the physical structure of the data

SECTION 9

MIL-STD-2168 AND QUALITY ASSURANCE

This section addresses the new standard for controlling the quality of software development, DoD-STD-2168. This standard is meant to be used in coordination with DoD-STD-2167.

9.1 Overview

9.1.1 Application and Definitions

DoD-STD-2168 is a draft standard (dated September 1986) for evaluating and controlling quality in the development and support of software. The standard is addressed specifically to Mission-Critical Computer Resources (MCCR) as defined in draft DoD Directive 5000.29. The standard may also be applied to non-MCCR software and to software used in the development, testing, and production of MCCR hardware.

The standard stresses two kinds of flexibility. The standard provides for flexibility in its own application through a process called "tailoring" the standard. DoD-STD-2168 also promotes flexibility in the precise organization, methodologies, and standards which will be used on each program to ensure quality. These are defined by the contractor, with DoD-STD-2168 defining only an overall approach. DoD-STD-2168 also defines criteria for measuring the contractor's plans and performance with respect to that approach.

DoD-STD-2168 requires the contractor to plan for and build in software quality as called for in DoD-STD-2167. In support of that effort, the contractor should establish and implement a software quality program. This program should:

- Evaluate the requirements established for the software
- Evaluate the software development methodologies, as planned and as implemented
- Evaluate the resulting software and documentation

- Provide feedback based on the evaluations and perform the follow-up actions necessary to ensure that all identified problems are resolved

It is intended that this fundamental approach be applied over a wide range of environments and program scopes. The standard is broadly applicable, because of the tailoring process and because of the "hands-off" approach toward the organization and detailed methods that the contractor uses to implement the fundamental approach.

9.1.2 Evaluation Activities

DoD-STD-2168 calls for two distinct types of evaluation. The first is evaluation of the activities by which the software is produced. The second is evaluation of the products of those activities, including the final product, the software itself. Both kinds of evaluation are discussed in general terms in Section 4 of the standard. After discussing the establishment of a plan, qualifications of evaluators, and plan development, the section discusses several specific evaluations that are to be repeated in each phase of the program. The following are evaluated:

- Compliance with requirements
- Tools and facilities
- Software configuration management
- The software development library
- Documentation and media distribution
- Storage and handling
- Risk management
- Subcontractor products
- Commercially available reusable software and Government-furnished software

While DoD-STD-2168 calls for these evaluations to be conducted, the actual performance of the activities evaluated is covered by other standards. The tailoring process may delete some of the evaluations, notably subcontractor-product evaluation, for any given program.

In addition to the evaluations, Section 4 spells out several activities performed to administer the software evaluation effort itself. The concerns that must be addressed include, among

others, quality records, corrective-action processes, quality reports, and interfaces with other organizations.

Section 5 of DoD-STD-2168 deals with the stages of the program. For each stage, the activities enumerated in Section 4 are discussed together with evaluation steps specific to the activities of the stage. In addition, Section 5 specifies the products to be evaluated and the reviews to be carried out within each stage. This separation of the evaluation of products from the evaluation of the activities producing the products is central to the approach adopted by DoD-STD-2168.

9.2 Management Guidelines

9.2.1 Processes Not Explicitly Covered

DoD-STD-2168 is broad in its coverage of both products and the activities producing the products. Certain topics, however, are not addressed:

- The standard carefully avoids specifying how the contractor is to carry out the overall approach. In every case, the exact methods, personnel, and organization are left for the contractor plan. This promotes flexibility. It is, of course, the Government's responsibility to evaluate the adequacy and cost-effectiveness of each contractor's plan for meeting the needs of each program.

- DoD-STD-2168 calls for technical planning but does not address management planning. None of the DoD-STD-2168 deliverables covers review of budgets, schedules, or manning plans, for example. Since such management plans can affect the probability of quality software being produced, it is the task of the program manager to review such issues independently of the procedures specified by DoD-STD-2168.

9.2.2 Tailoring Guidelines

Since DoD-STD-2168 is intended to cover a wide range of programs, it is necessary to vary the requirements that it imposes. This process is called tailoring. Tailoring consists of removing some of the requirements in the standard, often on a product-by-product basis. The goal is to ensure that quality software is achieved without incurring excessive costs. Guidelines for tailoring are provided by Appendix D of DoD-STD-2168. Detailed guidance on tailoring will be contained in DoD-HDBK-286, which is planned for future release.

The PMO tailors DoD-STD-2168 while preparing an RFP. In tailoring DoD-STD-2168, the PMO must maintain consistency with the SOW and with other standards invoked. For example, if a phase or deliverable is tailored out of DoD-STD-2167 for a given program, the same phase or deliverable must be tailored out of DoD-STD-2168.

Contractor input to the tailoring process may be helpful. Such input can be solicited by issuing draft RFP's. Even after an RFP has been released, contractors may offer proposals suggesting a tailoring different from that in the RFP.

9.2.3 Relationship to DOD-STD-2167

Section 5.8 of DoD-STD-2167 provides a summary of the approach spelled out more completely by DoD-STD-2168. In addition, DoD-STD-2167 specifies a Software Development Plan (SDP) which may, for a small project, provide the only required definition of plans for software quality evaluation. DoD-STD-2168 may be considered as an expansion of and replacement for Section 5.8 of DoD-STD-2167. That section should be tailored out of DoD-STD-2167 whenever DoD-STD-2168 is applied.

Because DoD-STD-2168 was developed after DoD-STD-2167, the concepts and DID's defined by DoD-STD-2167 were available to the writers of DoD-STD-2168 and were incorporated into DoD-STD-2168. While the two standards are closely related and intended for coordinated use, the use of one does not automatically require the use of the other. DoD-STD-2168 may be used in conjunction with any software development standard.

SECTION 10

GUIDELINES FOR PROPOSAL EVALUATION

This section presents guidelines for evaluation of proposals that require the use of Ada as the implementation language. It should be used in conjunction with other evaluation criteria, as it is limited in scope to issues involving Ada.

10.1 Understanding the Problem

The offerer's proposal should demonstrate an understanding of issues related to the use of Ada. These include technically appropriate application of Ada, recognition of the risks entailed in using Ada, and compatibility with Air Force policy.

The offerer's proposal should include a clear understanding of the appropriate use of Ada with regard to the software development in general and the application in specific. The offerer should demonstrate an understanding of how Ada language features support the software-engineering principles of modularization, information hiding and encapsulation, and how these features apply to the particular application. If the level of detail required by the sponsoring agency includes the preliminary design of the system, then the offerer should identify the major Ada packages that will comprise each CPCI along with the rationale for the decomposition selected. The major data structures, procedures, and tasks should be identified and defined within these package specifications.

The offerer must also identify any risks associated with the use of the Ada language. Any limitations imposed by the run-time environment, timing requirements, and capacity constraints should be identified as risk areas. Issues such as compiler reliability should also be addressed. A proposal might describe special support arrangements with the compiler vendor or other contingency plans to be invoked in case a compiler should prove to be unreliable.

The offerer should also demonstrate understanding of the current Air Force and AJPO policy regarding the use of Ada. Policy issues may include, for example, the use of validated and project-validated compilers (see Section 2.2.2), the use of approved environments, and parallel development in another language to minimize risk. The offerer should propose to use products that are compliant with this policy.

10.2 Personnel and Training

Qualified personnel are extremely important to the success of Ada software development efforts. The offerer should identify key Ada personnel who will be assigned to the project. The offerer should indicate its commitment of these key people within the proposal and no substitutions should be permitted without government approval. Ideally, these key persons will be experienced in the application, the Ada language, the proposed environment, and the proposed methodology.

The role of each individual within a project should be commensurate with his or her experience. In general, design and test responsibilities should be assigned to senior personnel, especially when multitasking is involved. Each project have at least one knowledgeable individual (an "Ada guru") responsible for addressing Ada language issues. Depending on the size of the project, this may be a part-time position or a position shared by several individuals. The offerer should stipulate that the individuals identified as Ada language experts will remain assigned to the project for the entire software life cycle.

In the near term, it is unlikely that an offerer can propose experienced Ada personnel for each labor category. The offerer should be required to provide an Ada training plan stating how both its entry-level and experienced personnel will be trained. Training should cover the following areas:

- The Ada language
- The underlying software-engineering principles that must be understood to use the Ada language effectively
- The use of the proposed software-development environment

The plan should indicate the type of training (e.g., lecture, hands-on use of Ada, or computer-aided instruction), the duration of the training, and topics covered. The Ada training plan should identify the provider of training and the provider's qualifications. The offerer should detail the informal training materials (e.g., video tapes and books) available to the project team. All entry-level programmers should be required to have formal training in both the Ada language and underlying software-engineering principles before being assigned to programming efforts.

Ada qualifications are relevant for management as well as technical personnel. Program managers need to understand how Ada impacts project schedules and costs. Proposed program managers should have experience managing Ada projects. In the near term, due to the novelty of Ada, this may be an unrealistic expectation. However, when proposed managers lack Ada experience, proposals should describe provisions to educate management.

10.3 Related Corporate Experience

It is obviously essential that the offerer have experience with the application for which the RFP has been issued. However, the offerer should provide a detailed description of its Ada-related experience as well as its application experience. The strongest related corporate experience will be demonstrated by an organization that has written a similar application in Ada.

In the near term, it will be common for organizations that have application experience to be void of Ada experience. Application experience alone does not qualify such an organization, since the lack of Ada experience can lead to overruns in schedule and cost. An offerer without Ada experience may propose to augment its team by adding a subcontractor with Ada expertise. Such proposals should include a management plan explaining how the two organizations will synergize their expertise.

10.4 Implementation Selection

In cases where the offerer is free to select a compiler and environment, there are several considerations that can impact the project. Ada environments vary from those with the minimum set of tools required to compile, link, and execute programs to sophisticated environments supporting multiple targets, configuration management, static and dynamic analysis, documentation, and testing. The proposal should be explicit about what tools are in the environment, what their functions are, and how they will support the effort. Specifically, the offerer should state:

- How the proposed environment supports the proposed methodology
- If the target is different from the host, facilities for interactive debugging and real time monitoring of software execution
- The degree to which the tools are integrated
- The degree to which the environment can support both experts and beginners
- What system resources are required (including CPU, memory, and peripherals)
- How efficiently the environment executes
- How the environment is being maintained
- The proprietary nature of the tools

It is especially important for the offerer to reveal proprietary restrictions on tools, since such restrictions can complicate future enhancements and modifications and make them more expensive.

10.5 Feasibility of Schedule

In asserting the feasibility of the technical proposal, the offerer should state the method used to arrive at the proposed schedule. The offerer should present a full understanding of the areas of risk associated with the schedule. If a model such as COCOMO was used (see Section 12), to what extent did the offerer consider the impact of Ada? How accurate has the model been for other efforts?

SECTION 11

REUSABILITY AND PORTABILITY

A principal goal of the Ada language is to reduce software development costs. This goal can be achieved in part by reusing software that has already been written and by porting working software from one execution environment to another. Indeed, since demands for new software are growing more quickly than the supply of programmers [Lie86], such techniques may soon become a practical necessity.

Reusability and portability are closely related concepts, but they are not synonymous. Section 11.1 defines each term and explains the distinction. The Ada language does much to allow and encourage the writing of reusable and portable software, but Ada programs are not automatically portable or reusable. Section 11.2 addresses management, design, and coding techniques that enhance reusability. Section 11.3 addresses management, design, and coding techniques that enhance portability.

11.1 Definition of Reusability and Portability

A computing system is often best understood as a set of hardware and software layers. Each layer provides mechanisms, and mechanisms at lower layers are used to implement mechanisms at higher layers. For example, one might identify the following sequence of progressively higher layers:

- integrated circuits
- a microarchitecture implemented in terms of integrated circuits
- microcode implementing a higher-level architecture in terms of the microarchitecture
- an operating system kernel
- operating system file management services
- system utilities like database management and configuration management
- low-level utility subprograms within a main program

- higher-level subprograms within a main program
- a main program
- an application system consisting of many main programs

Sometimes the mechanisms found in one context can be used with little or no modification in another context. Reusability is the ease with which a given low-level mechanism can be used to implement a wide variety of useful higher-level mechanisms. Portability is the ease with which a given high-level mechanism can be implemented on top of a wide variety of lower-level mechanisms. For example, a graphics package is reusable if scales, axis labels, and so forth are user-specified rather than "hard-wired" by the program for a particular application; the package is portable if it can easily be modified to generate commands for a variety of plotting devices.

Software cannot be classified absolutely as reusable or unreusable, or as portable or unportable. There are degrees of reusability and degrees of portability, corresponding to the effort required to make a mechanism work in a new context. In the ideal case, existing software can be used without modification for a new purpose or with new underlying mechanisms. As levels of reusability or portability decrease, the required effort increases, until a point is reached where the effort required to reuse or port a mechanism is greater than the effort required to reimplement the mechanism from scratch.

Reusability and portability are not always appropriate goals. Certain software can only be made reusable or portable by making it more complicated or less efficient. Therefore, the emphasis that a program manager places on reusability and portability must be considered in the context of other program goals and requirements.

11.2 Reusability

11.2.1 Classification of Reusable Software

The term reuse has been associated with a wide variety of techniques, ranging from the direct incorporation of a catalogued subprogram to the manual analysis of one main program's design to help in writing another main program. Our definition is broad enough to incorporate all such techniques. To distinguish among the various techniques, reusable software can be classified according to two attributes, the size and scope of the software part being reused and the kind and amount of modification necessary to reuse the part.

The Size and Scope of a Reusable Part

Software parts of different size and scope may be reused:

- General-purpose subprograms and packages (for example, a list-manipulation package) may be reused.

- Code specific to a given application area (for example, signal processing or missile guidance) may be reused for a number of related applications. The Common Ada Missile Packages (CAMP) effort sponsored by the Air Force Armament Laboratory [A&M85] is a noteworthy example of this approach. Over 200 software parts are being designed specifically for reuse. These parts include both general-purpose parts and parts providing specific armonics (armament electronics) facilities (for example, application-specific data types and constants, equipment interfaces, navigation operations, guidance operations, mission control and sequencing, warhead control, and telemetry). Ten missiles were studied to identify common features. To test the reusability of the CAMP parts, they are to be used to implement a control program for an eleventh missile, not part of the original study.

- A main program (for example, a screen editor) may be reused in a number of systems.

- The design of one program may be reused, with modifications, to write a similar program. For example, the top-level design of a program to maintain a data base on fuel supplies might be used as the basis for the top-level design of a program to maintain a data base on supplies of spare parts. As the spare-part program's design is refined, a detailed design distinct from that of the fuel-supply program will emerge. Certain portions of the spare-part program's code might be taken with little or no modification from tne fuel-supply program, while other portions might be written from scratch.

The Effort Required to Reuse a Part

The second attribute of software reuse is the amount of effort necessary to reuse the part. This effort can be measured in terms of the amount of text that must be modified and the amount of analysis that must be performed to achieve tnat modification. Mechanical modifications can be facilitated by text editors or specialized tools, but more sophisticated modifications must be performed by hand after careful thought. There is a wide range of possibilities:

- Some software may be directly reusable, without modification. If the software is a separate Ada compilation

unit, it can be compiled into the program library of the reusing program, or even linked into the reusing program from a common library without recompilation. Otherwise, the software can be extracted from one file and inserted in another file using a text editor.

- An Ada generic unit can be reused by writing a generic instantiation. A typical generic instantiation is only a few lines long and consists of the stipulation of a small number of specific values, variables, subtypes or subprograms as generic actual parameters. The result is an instance of the generic unit tailored to solving the problem at hand.

- If the part to be reused was not originally written as a generic unit, the effect of creating a slightly different instance can be achieved by systematically editing a copy of the part. In some cases this may involve little more than a command to replace all occurrences of one word by another word. Mechanical substitution may be necessary because the original writer consciously or unconsciously neglected to make the unit generic, or because the variations needed could not be reflected in a straightforward way by generic parameters. For example, there are no generic parameters for exceptions, so editing is required to create a new package identical to a previously existing package except for the exceptions it raises.

  When generic parameters are adequate to reflect the differences in the old and new versions, the new version can often be written as a generic unit with little additional effort. Then, with proper configuration management, the old version can be replaced in subsequent releases of the old program by an instantiation of the new generic unit. From that point on, there will be only one version to maintain, and any further reuse in other programs will be made easier.

- Sometimes parts can be reused by editing them in more sopnisticated ways. Despite substantial differences between the old and new programs, there may be substantial blocks of code that can, after careful analysis, be extracted from the old program and incorporated in the new one. Rather than applying mechanical substitutions, one "cuts and pastes," interspersing new code with old code. For example, a statistical procedure to normalize a two-dimensional array might be modified to normalize a three-dimensional array by replacing two nested loops with three and extending all array-component references with a third index value. The code for a simple compiler that does not produce a cross-reference listing could be reuced to write a cross-reference tool by replacing calls on code-generation procedures with code to add information to a cross-reference table. Though a large amount of new coding will be required, there will also be substantial savings.

- As noted earlier, there may be cases in which a high-level design is reusable. In this case, the effort to reuse is substantial, including a new detailed design effort. There are preliminary research efforts, notably the European PROSPECTRA project [Kri86], concerned with automating the transition from formal specifications to designs to code. Code would be derived from designs and designs from formal specifications by sophisticated pattern-driven transformations known to preserve the meaning of the text being transformed. Reuse of a design would consist of "replaying" the sequence of transformations used to derive the old program from the design, replacing certain old transformations with new ones as appropriate. Rather than reusing code, this approach reuses the decisions made during the coding process. While this approach is promising in the long term, much work remains to be done before it can be proven feasible and made practical.

Whenever software not originally intended for reuse is used in a new context, there is a risk that the new context will violate some subtle assumption upon which the software relies. The greater the amount of modification that is necessary to adapt software to a new use, the greater this risk.

11.2.2 Management Approaches to Promote Reusability

Software intended for reuse is generally more complex than software intended for a single use. When writing single-use software, a programmer can exploit special properties of the problem at hand and the context in which the software will be used. A programmer writing reusable software must be able to accommodate a wide range of possible uses, and fewer specific properties of the problem may be assumed. The resulting additional complexity tends to make reusable software more expensive to create and maintain than single-use software.

Software intended for reuse is often less efficient than single-use software. Reusability often results from generality, and generality often results from determining certain information at execution time rather than at the time a program is written. Furthermore, reusable software must account for certain extreme values and exceptional conditions that might be known not to occur in a particular application.

Thus reusability comes at a price. Often the price is worth paying, but a program manager must evaluate the risks and costs of reusable software. Reusable software presents managers with two problems -- reusing existing reusable parts and creating new reusable parts.

When potentially reusable software already exists, the manager must decide which, if any, software to reuse. The quality of the existing software must be high enough to justify its

reuse. Quality includes such attributes as reliability, efficiency, and clarity. Furthermore, the effort involved in adapting the existing software to its new use must be substantially less than the effort of building new software from scratch. To keep options as wide as possible, program managers should encourage the development of reusable software, both inside of their organizations and elsewhere. Because of the disincentives of higher risk, higher cost, and lower efficiency, positive incentives for developing reusable software are required. This problem is addressed in Section 11.2.5 below.

When software adaptable to the purpose at hand does not already exist, the manager must decide whether to produce single-use or reusable software. The scope of the problem to be solved by the reusable component must be selected judiciously. Every problem is a special case of another, more complex problem. A certain degree of generality can often be achieved with minimal impact on complexity and efficiency, but overgeneralization can impose time and complexity costs in return for capabilities that may never be needed.

11.2.3 Design Approaches to Promote Reusability

Reusability is enhanced by making a software part applicable to a wide variety of uses. There are many design measures that can be taken to increase a part's generality. These include design for portability, the use of standards, the decomposition of a program into parts with narrowly defined functions, the use of simple and explicit interfaces, and parametric abstraction.

Because a portable part is usable with a wide variety of underlying mechanisms, it has more uses than a part that is not portable. Thus portability is an important factor contributing to reusability. Measures to enhance portability are described in detail in Section 11.3.

Standardization of entities outside of software can enhance software reusability. If individual programs define their own input and output data formats, the subprograms used to parse input data and format output data are not likely to be reusable elsewhere. However, if a common format for input and output data is adopted for a group of programs, a common set of input and output tools can be shared by all programs in the group. Not only the subprograms, but the data files themselves, may then become useful for a wider variety of purposes. Similar considerations apply to communications protocols, error-reporting methods, and many other aspects of system design.

The principle of functional cohesion can enhance reusability. If the various functions performed by a program are isolated in different modules, each with a narrowly defined purpose, then it is more likely that individual modules will be found useful in other programs. If distinct functions are

combined in a single module, that module will only be useful in other applications requiring all of those functions in combination. For example, a function to sort aircraft identifiers by the projected round-trip range of each aircraft, given its current fuel supply, is very narrowly applicable. If, instead, we write a function to compute projected ranges from fuel supplies and a sorting procedure, both the range function and the sorting procedure are likely to find use elsewhere.

Even more important to reusability is the design principle of weak coupling. Modules should not interact except in a few well-understood ways. If modules communicate by setting and examining variables external to one of the modules, each module is highly dependent on the context in which it occurs. It will be difficult to extract one of the modules and reuse it in a different context. If, instead, modules interact only by passing parameters, then the modules have explicit, easily understood interfaces. Such interfaces can be understood independently of the context in which they are used in a particular program.

A related principle is parametric abstraction. Though a module may only need to deal with certain values for the current application, the module's algorithms may be just as easily applicable to other values. In such cases, the module can be made more general by specifying those values as parameters. Generic parameters will be appropriate in some cases and subprogram parameters will be appropriate in others. For example, in a function to determine the time of sunset in Washington, D.C., on a given day of the year, the latitude and longitude of Washington, D.C., may be coded into the algoritnm as constants. If parameters are used instead, a more generally applicable function, for determining the time of sunset for any given city and day, will result. Similarly, a procedure to read from the default input file or write to the default output file can be made more general by specifying the file as a parameter.

Besides the problem of making a part reusable, a designer faces the problem of finding reusable parts that already exist. A huge collection of reusable parts is of no value unless parts applicable to a given purpose can be found in the collection. The difficulty of this information-retrieval problem increases as efforts to produce reusable parts succeed. One approach to solving this problem is to establish a taxonomy for classifying reusable parts. Another is to build tools to find the required parts. Efforts to construct such a tool are still in the research stage. [A&M85] describes a proposed expert system for finding appropriate reusable parts. [Coh86] describes an approach based on comparing the formal specifications of a needed part with the formal specifications of the reusable parts in the catalogue, searching for a match.

11.2.4 Programming Approaches to Promote Reusability

Programmers can make code easier to reuse by minimizing explicit references to specific values. For example, in a loop meant to iterate over each component of an array, one can write

```
for I in A'Range loop
```

rather than

```
for I in 1 .. 10 loop,
```

so that the loop will still perform as desired if the bounds of the array A later change. Other attributes, including the 'First, 'Last, 'Delta, and 'Digits attributes, can be used in a similar way to ensure that a change to an object or type declaration will not require changes to an algorithm. By avoiding reference to specific values, a programmer makes code reusable in applications with different values.

When stipulation of a specific value is unavoidable, a constant or named-number declaration should be used to give the value a symbolic name. Subsequent references should be to the symbolic name rather than to the actual value. Code can be reused with different values simply by updating the constant or named-number declaration. This approach is similar to parameterization. While the use of generic or subprogram parameters may incur a performance penalty with some implementations, the use of symbolic names will not. However, the use of symbolic names requires program text to be modified before it is reused. This entails not only the effort to modify the text, but the maintenance and configuration management of multiple versions of a module.

Reuse of a part requires knowledge of how a part is to be invoked, what its intended effects are, and restrictions on its use. Thus, to be reusable, a part must be thoroughly documented. The documentation must be sufficient for the part to be usable by people unacquainted with the project and programmer that produced it.

Since portability makes a module more generally applicable, reusability will be enhanced by following coding guidelines designed to enhance portability [N&W84, Sof84]. Portable programming is addressed in Section 11.3.

11.2.5 Encouraging the Production of Reusable Software

The organizations in the best position to produce reusable software are not always the organizations most likely to benefit

from its reuse. Organizations must be given incentives to produce reusable software despite the potentially higher risks and costs. We consider four potential sources of reusable software for Air Force programs:

- o Libraries maintained by the Air Force project
- o Libraries maintained by software contractors
- o Proprietary packages
- o Libraries in the public domain

In some cases, existing organizational, economic, and legal systems work to discourage reuse of software. We consider not only incentives within the current systems, but also changes to these systems that would encourage the development of reusable software.

Project Libraries

In the ideal case, project planners have enough foresight to recognize that a particular facility, or family of related facilities, will be needed at several places within a single project. Reusability can then be mandated as one of the requirements for a software part providing that facility. The anticipated needs within the project determine the appropriate degree of generality for the part.

Often, however, opportunities for reusability do not become evident until later in the project. The greater the number of organizations working independently on different parts of the project, the smaller the likelihood that opportunities to reuse the same part will be recognized. This effect can be counteracted by a matrix organization in which managers responsible for different subsystems contract internally to a central programming staff for the production of software parts. If the central programming staff is motivated to minimize its costs, this will provide an incentive for the staff to seek and exploit opportunities for reusability.

Individual programmers should be rewarded for writing software that is written and documented in a manner conducive to reuse. Often the pride of knowing that one's work has been selected for reuse is a reward in itself. However, more tangible rewards, such as public recognition, interesting follow-up assignments, and positive performance evaluations are likely to be more effective.

A program manager's primary responsibility is the successful completion of his own project. Given a choice between writing a low-risk, low-cost single-use part that will benefit one's own program or a higher-risk, higher-cost reusable part that will

benefit many programs, the manager can most effectively fulfill his responsibility by selecting the first alternative, even though Defense Department programs collectively would benefit more if all program managers chose the second alternative. This problem could be overcome by budgeting mechanisms that allowed one Defense Department program to "sell" a reusable part to another Defense Department program. The managers of the two programs would agree upon a cost, which would be deducted from the budget of the "buying" program and added to the budget of the "selling" program, in compensation for the costs incurred to make the part reusable. Alternatively, an organization could pay a fee to a central Defense Department repository for the use of a reusable component, and the organization that provided the component would then receive a royalty.

Contractor Libraries

Competitive bidding encourages potential contractors to seek the lowest-cost solution to a requisitioner's problem. If a part can be reused several places within the software product, thereby reducing the cost of the project, development of a reusable part is likely to be proposed. However, contractors are unlikely to propose the development of a part conducive to reuse on other projects, because this is likely to raise the cost of developing the part and is not necessary to meet the requirements of the current solicitation. If a program manager believes that part of the product to developed under a given contract may be usable elsewhere in the program, specific requirements for reusability should be included in the solicitation.

In the long run, contractors would benefit by developing their own in-house libraries of reusable parts. By proposing incorporation of these already-developed parts, bidders could be more competitive, offering lower costs, lower schedule risks, and greater assurances of reliability. In the short run, however, their are several disincentives to the development of contractor libraries:

- Software is not recognized as capital by current accounting practices or tax laws. Development of software shows up in financial statements as an expense rather than as an exchange of cash for other assets. Though it may be rendered obsolete, software may not be depreciated.

- Software development companies are generally service-oriented, deriving income primarily by billing customers for the labor of the company's employees. While labor billable to a contract adds to the company's profits, labor that is not directly billable to a contract increases the company's overhead and makes its bids less competitive. When the company provides a service, it is immediately reimbursed for the expenses it incurs. In contrast, if the company were to develop reusable software without having identified a

specific customer, it would assume the risk of not recouping its expenses. This risk is an integral part of a product-oriented business, but not a service-oriented one.

- Typical Defense Department contracts include data-rights clauses providing the Government with ownership of all delivered software. This discourages a contractor from incorporating proprietary reusable software in its delivered products and raises the question of how much the contractor can charge the Government for a later software product incorporating the same reusable part.

These disincentives are, for the most part, beyond the control of the Air Force program manager. However, there are steps a program manager can take to help encourage the development of contractor libraries. Requests for proposals can emphasize the importance of an on-hand supply of reusable software parts and data rights clauses can be written to protect the contractor's investment in reusable software.

In requests for proposals, the program manager can explicitly list the availability of already-developed reusable parts as an evaluation criterion. The increase in technical competitiveness and credibility resulting from the availability of reusable parts will help compensate for the decrease in cost competitiveness resulting from the overhead of their development. Indeed, if contracts are lost because of the lack of on-hand reusable parts, contractors will have little choice but to develop such parts.

Data-rights clauses can be written to recognize explicitly the role of contractor-owned reusable software. The contractor should be permitted to include in the delivered product software parts whose development costs have not been charged to the contract. Despite such inclusion, the contractor would retain full rights to any such software part, including copyright. Should the Government choose to derive a new software product from one that includes contractor-owned reusable parts, the right to reuse these parts would have to be purchased from the contractor. (The situation is analogous to the contractor purchasing a proprietary part from a third party and including it in the delivered software.) Since the Government did not pay for the original development of these parts, the net cost is likely to be lower. Furthermore, the existence of contractor-owned reusable parts may lead to lower-cost bids on other proposals.

Proprietary Libraries

One of the goals of Ada's designers was to provide a foundation for a software parts industry. This product-oriented industry would produce reusable software parts that could be purchased off-the-shelf and combined to build large programs,

just as semiconductor chips are purchased off the shelf and combined to build powerful circuit boards.

The software-part industry is still in its infancy. Two early entrants, EVB Software Engineering's GRACE and Quantitative Technology Corporation's Math Advantage Library, are discussed in Section 14. As the Ada parts industry matures, we can expect it to improve in a number of ways. Parts will be designed to exploit the capabilities of the Ada language and to be consistent with accepted Ada style conventions. Larger-scale parts will be built, providing solutions to application-specific problems of significant complexity. Field experience will improve the reliability of the parts. Wider competition will lead to the sale of individual parts or groups of related parts at reasonable prices.

Once it reaches maturity, this industry is likely to become an important source of reusable Ada components. Program managers can encourage this maturation. Companies selling reusable Ada parts should be made aware of Air Force program needs. Public announcements of intent to purchase a particular kind of reusable part may stimulate the production of such parts. In special cases, Air Force programs might support the industry by soliciting competitive bids for the production of specific reusable parts at Government expense, with the Government acquiring full data rights.

Public-Domain Libraries

Some reusable Ada parts are already available in the public domain. A software pool like the Ada Repository discussed in Section 14 can be an effective vehicle for the sharing of Government-owned reusable parts among various Government organizations (including NASA as well as the Defense Department), subject to the caveat that such software is generally unsupported and used at the borrower's own risk. Universities, corporations, and individuals, however, are less likely to donate software to such pools as the financial incentives for selling Ada reusable parts increase. Judging from the precedent of microcomputer "freeware," most donations are likely to be from authors who have not invested the effort necessary to make the software a market-quality product.

11.3 Portability

11.3.1 Definition of Equivalent Behavior

There are several aspects of a program's behavior that are not fixed by the rules of the Ada language, and so may vary from one implementation to another. Language rules forbid dependence on certain implementation-dependent aspects of a program's behavior, such as the way subprogram parameters are passed and the order in which parts of an expression are evaluated, though such rules are practically unenforceable with the current state of compiler technology. Dependence on other implementation-dependent aspects, including task scheduling and numeric precision, is allowed but is obviously detrimental to portability.

Because of these variations, there may be observable differences in a program's behavior under different implementations. A program can sometimes be considered portable in spite of these differences. Portability was defined earlier as the ease with which a given high-level mechanism can be implemented on top of a wide variety of lower-level mechanisms. Implicit in this definition is the assumption that the high-level mechanism will behave equivalently in each case. It is important to distinguish, however, between equivalent behavior and identical behavior.

A program's behavior has many attributes, only some of which are of interest to the program's users. If the attributes of interest are identical under different implementations, the behaviors under those implementations are acceptably equivalent and the program is portable. Consider the following examples:

- If a program without real-time constraints produces the same outputs under different implementations, but in different amounts of time, the program is portable.

- A function designed to compute cosines to five significant digits may be portable even if results differ in their sixth digit under different implementations.

- A factorial function may work for arguments up to 10 under one implementation and up to 20 under another implementation. If arguments higher than 10 never arise in a particular program using the function, the program may be portable.

- A procedure to find values of X and Y that minimize the value of F(X,Y) for some predetermined function F may be implemented in terms of a multitasking search. If F has more than one point at which it reaches its minimum value,

different implementations may produce different, but equally acceptable, results. The procedure should be considered portable.

This view of portability is largely subjective. Portability cannot be judged only by objective observation of a program's behavior. Users' expectations must also be considered.

11.3.2 Impediments to Portability

There are many reasons a program can fail to be portable:

- The program's purpose may be inherently machine-dependent.
- The program may have machine-dependent hardware interfaces.
- The program may use implementation-defined features of the Ada language.
- Programming style may introduce implicit implementation dependencies.

This section considers each of these situations in turn.

An operating system is an example of a program whose purpose is inherently machine-dependent. The purpose of an operating system is to manage computer resources and use those resources to provide services to computer users. When the purpose of a program is inherently machine-dependent, we can expect the central algorithms of the program to be machine-dependent. Since the program is intimately connected to the target machine, it makes no sense to consider running the same program on a different target machine. Thus portability is not a reasonable goal for such programs.

Consider a computer-aided design program that uses a specialized graphics terminal for data entry and display. The subprograms to query and drive the terminal may be highly machine-dependent, but most of the internal processing may be machine-independent. If we view the terminal interface as a low-level mechanism and the remainder of the program as a high-level mechanism implemented on top of this lower-level mechanism, then the high-level mechanism can be written in such a way that it is portable. To move the program to another target computer with different terminals, it would then only be necessary to implement new low-level mechanisms.

The Ada language contains certain features that are implementation-defined, meaning that each implementation determines the allowable forms for that implementation and the exact meaning for each form. Examples of implementation-defined features are predefined numeric types like LONG_INTEGER,

LONG_FLOAT, SHORT_INTEGER, and SHORT_FLOAT; representation specifications; attributes other than those defined in Annex A of the Ada Language Reference Manual; and machine-code procedures. A compilation unit using such features may be compilable on some Ada compilers but generate compile-time error messages on others. Even if the unit is successfully compiled on two different compilers, the meaning of the features may be different in each case. A compilation unit containing implementation-defined features is inherently unportable. However, if such features are found only in the lower-level mechanisms of a program, the higher-level mechanisms of the program may still be portable.

Implementation-defined features must be distinguished from implementation-dependent features. The form and meaning of implementation-dependent features are defined in the Ada Language Reference Manual, but the meaning is defined loosely enough to permit significant variation from one Ada implementation to another. The most subtle impediment to portability in an Ada program is reliance on a particular behavior for a feature whose behavior is implementation-dependent. Such reliance often results from subconscious assumptions by the programmer. The program works correctly on the original target machine and is likely to compile correctly on other compilers. However, the program may behave differently on other targets, and the differences may go beyond acceptably equivalent behavior, as defined in Section 11.3.1. Worse yet, the differences may only arise in unusual situations, so a program may appear to have been ported successfully, then fail unexpectedly in the field.

11.3.3 Promoting Portability

The Ada Language Reference Manual defines the meaning of implementation-dependent features so loosely that the construction of portable software is practically impossible without making some minimal assumptions about the implementation. For example, an Ada implementation could theoretically conform to the language rules even if an attempt to invoke the main program always raised the exception STORAGE_ERROR or if the allowable range of integers were only -1 to 1; but it is hard to imagine a useful program that could be ported to such an implementation. Fortunately, it is easy to compose a realistic list of minimal assumed capabilities for all implementations to which a program might be ported. These capabilities are supported by most or all currently validated compilers and are adequate to implement a wide variety of useful software. Such lists are found in [N&W84] and [Sof84].

Thus the notion of portability is relative for three different reasons:

- As explained in Section 11.3.1, portability can only be measured in terms of an agreed-upon definition of acceptably equivalent behavior.

- As explained in Section 11.3.2, even if a program's low-level mechanisms are not portable, higher-level mechanisms may be.

- As explained just above, "portable" software is generally only portable to a class of implementations providing certain minimal capabilities.

Because portability is relative, it makes more sense to speak of promoting portability than ensuring portability.

The Analyst's Role

If portability is an important requirement for a program, it is the analyst's role to describe this requirement precisely. The analyst must define acceptably equivalent behavior and state the minimum capabilities that may be assumed for targets to which the program will be ported. To the extent possible without constraining the program's design, the analyst should state which parts of the program should be portable without modification and which may require rewriting.

The Designer's Role

Designers can promote portability by isolating dependencies in separate modules. These modules can then be viewed as low-level mechanisms upon which the rest of the program is implemented. The dependencies that should be isolated in this way include machine-dependent interfaces and all uses of implementation-defined features.

Three levels of mechanisms enter into this isolation. The highest, or logical level, consists of machine-independent processing. The lowest, or physical level, consists of underlying mechanisms such as a bare machine, a device, or an operating system. Porting a program means making it run on top of a new physical level. Between the logical and physical levels is the interface level. The interface level uses the physical-level mechanisms to implement abstract mechanisms used by the logical level. Dependencies should be isolated in the interface level.

The interface level will typically consist of packages whose specifications describe abstract services provided to the logical level. These specifications should not in any way reflect the nature of the physical level. An interface-level package specification will remain fixed when the program is ported, as will all logical-level units naming that package in a with clause. The corresponding package body, stipulating the implementation of interface-level mechanisms in terms of

physical-level mechanisms, must be rewritten to use the new physical-level mechanisms.

The Programmer's Role

Features with implementation-dependent behavior pervade the Ada language, so the use of these features cannot be avoided or isolated. However, programmers can promote portability by avoiding reliance on implementation-dependent aspects of a feature's behavior. This requires familiarity with the variations allowed among Ada implementations and the programming skill to ensure that acceptably equivalent results are obtained for all possible variations.

Implementation dependencies can be quite subtle, so fairly rigorous constraints must be followed to preserve portability. These constraints should be specified in programming guidelines that are enforced by management. Detailed programming guidelines specifically concerned with preserving portability are published in [N&W84] and [Sof84]. Research is underway into a tool that will automatically prove the portability of a program unit [Coh85], but this effort is still preliminary.

SECTION 12

ESTIMATING ADA DEVELOPMENT EFFORTS

This section includes a discussion of four existing software cost estimation models -- COCOMO, SLIM, Jensen's model, and PRICE-S -- and the use of these models for estimating Ada software development efforts. The discussion emphasizes the underlying assumptions of each model, how each of the models derives cost and scheduling data, and how each model might be affected by the Ada language. This section concludes by describing a workshop on the impact of Ada on COCOMO cost estimates for the WWMCCS Information System (WIS).

12.1 COCOMO

The COnstructive COst MOdel (COCOMO) was developed by Dr. Barry Boehm, Director of Software Research and Technology for TRW, Inc. Its purpose is to estimate the cost of software development efforts, measured in man-months (MM), and to estimate software development schedules, measured in months. Among the dozen or so major cost estimation models, COCOMO is the most completely documented and is non-proprietary. COCOMO can be applied to three different classes of software development projects, called organic, semidetached, and embedded. In addition, three versions of the COCOMO model, called basic, intermediate, and detailed, use successively more detailed information to derive more accurate predictions.

12.1.1 Underlying Assumptions of COCOMO

The primary cost driver of COCOMO is the number of delivered source instructions. This estimation must be made by some other process and is critical to the output of COCOMO. There are numerous ways to make such an estimate, including the use of historical data. Extensive databases about previous projects may contain such information as functional descriptions, implementation language, lines of code, complexity, and other attributes that can be used to estimate the size of a particular function. Delivered source instructions (DSI) do not normally include test drivers or other support software.

COCOMO cost estimates begin at the product-design phase and end at the completion of the integration and test phase. The

estimates do not include certain miscellaneous efforts that take place during development, such as user training, installation planning, and conversion planning. Only direct-engineering charges are considered; costs for computer operations, clerical support personnel departments, higher management and plant maintenance are excluded. The cost of project librarians and project management is included.

COCOMO cost estimates are based on certain assumptions:

- Prior to the product-design phase, thoroughly specified unambiguous requirements have been developed.
- The requirements are not changed substantially after the plans and requirements phases. Invariably, some requirements will be modified, and these normal changes are reflected in COCOMO.
- Good management practices minimize slack time.
- A man-month contains 152 working hours. (This figure accounts for average time off due to holiday leave, vacation and sick leave.)

Basic COCOMO and Intermediate COCOMO do not assume phase-dependent cost drivers, but Detailed COCOMO does.

12.1.2 COCOMO Development Modes

COCOMO partitions software development into three classes, or development modes, with common attributes affecting cost. The COCOMO model's computations are adjusted to account for these attributes. The three development modes are organic, semi-detached, and embedded. The partitioning factor is not size alone. The size of a software development effort, coupled with the mode, provide the necessary information for accurate forecasting.

The organic mode is characterized by small software teams with personnel experienced within the organization and experienced in the application. Organic mode projects require less communication in the early design, since project personnel are familiar with each other's capabilities and responsibilities. These projects are developed with flexibility towards meeting requirements. If a particular requirement cannot be met without disproportionate cost, then alternatives are relatively easy to negotiate with the users. Organic projects are rarely greater than 50,000 delivered source instructions (50 KDSI).

The embedded mode is characterized by tight constraints. The software usually must operate within a prescribed hardware environment with rigid operational procedures. The requirements are relatively inflexible. Examples are air traffic control and

electronic banking systems. In many cases, the application being developed is relatively unknown, requiring fewer, but more senior, analysts in the early stages. Large teams of programmers are brought on to do the actual coding and unit testing.

The semidetached mode of software development is somewhere between organic and embedded. The team members have a mixture of experienced and inexperienced personnel. The software to be developed has some characteristics of both organic and embedded modes. For example, if some part of the system has rigid specifications on communications interfaces, but flexible requirements on graphic display features, this system could be characterized as semidetached. Semidetached software can be as large as 300 KDSI.

12.1.3 COCOMO Computations

This section outlines the way in which COCOMO estimates are computed. Basic, Intermediate, and Detailed COCOMO use successively more detailed information and provide successively more accurate predictions. Details about the computations can be found in [Boe81].

The COCOMO estimating equations were obtained by analyzing 63 software projects. Figure 12-1 illustrates the variety of these projects. Boehm recognizes some deficiencies, such as too few microcomputer-based data points, but points out that the COCOMO estimates are accurate in each case.

Basic COCOMO estimates are based on two simple mathematical formulas. The first computes estimated effort in man-months, based on the number of delivered source instructions. The second computes estimated months to completion, based on the estimated effort. According to these formulas, estimated effort grows exponentially with respect to the number of source instructions and estimated time to completion grows exponentially with respect to the estimated effort. Both formulas contain constants whose values depend on the development mode.

Basic COCOMO gives only rough order-of-magnitude estimates. Boehm states that Basic COCOMO estimates are within a factor of two of the actual results only 60% of the time. Better approximations are achieved with Intermediate COCOMO, which considers several more detailed cost drivers.

Intermediate COCOMO estimates begin with the calculation of the nominal effort using the same formula as Basic COCOMO. The nominal effort is then multiplied by an effort adjustment factor, which is the product of fifteen cost-driver values. These values are based on attributes that have been widely accepted as influencing software development costs, and incorporated into a number of models. The fifteen factors are partitioned into four groups:

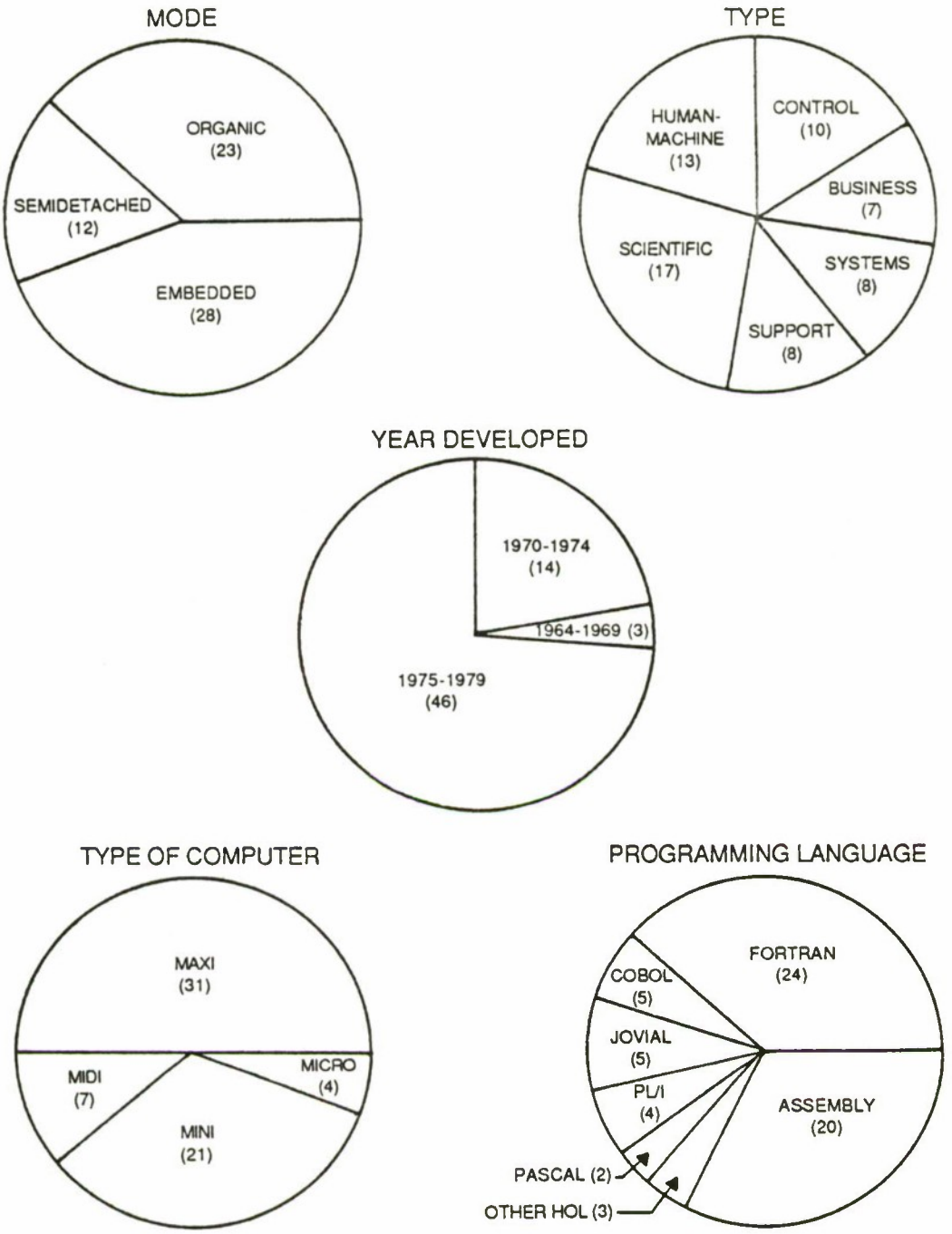

Figure 12-1. Attributes of software projects on wnich COCOMO is based. These charts break down the 63 COCOMO projects according to five different attributes.

- Product Attributes:
  - Required software reliability
  - Database size
  - Product complexity
- Computer Attributes:
  - Execution-time constraint
  - Main-storage constraint
  - Virtual-machine volatility
  - Computer turnaround time
- Personnel Attributes:
  - Analyst capability
  - Applications experience
  - Programmer capability
  - Virtual-machine experience
  - Programming-language experience
- Project Attributes:
  - Modern programming practices
  - Use of software tools
  - Required development schedule

Intermediate COCOMO development-time estimates are computed from effort estimates using the same formula as in Basic COCOMO.

Detailed COCOMO provides more accurate forecasts by using a three-level hierarchical decomposition of the software and by using phase-sensitive effort multipliers. The three-level decomposition is as follows:

- The lowest level, the module level, is described by the number of delivered source instructions in the module and by the programmer-capability, module-complexity, and programmer virtual-machine experience cost drivers.
- The second level, the subsystem level, is described by the remainder of the cost drivers. They are time and storage constraints, analyst capability, use of tools, and schedule constraints.

- The top level, tne system level, is used to apply major overall project relations such as the nominal effort and schedule equations.

12.1.4 Use of COCOMO to Estimate Ada Software Development Efforts

Many factors that will influence the development of Ada software have not been considered by the COCOMO estimation equations. Furthermore, because of lack of empirical data, some estimation equations may be skewed when applied to Ada. This section considers the impact of the Ada language on COCOMO.

The major driver of the COCOMO is number of lines of delivered source instructions. This number is hard to estimate for Ada programs, given the current lack of experience with major Ada development efforts. Some literature suggests that Ada is more verbose than other languages, which aids in readability but would distort estimation models. Because of Ada's unique module structure and tasking constructs, and the large amount of redundant information in Ada programs, it is not even clear that the number of source lines in an Ada program is directly proportional to the number of source lines in an equivalent program written in another language.

There appears to be general agreement in the Ada community that, when using Ada, more time will be spent earlier in the project life cycle, particularly in preliminary and detailed design. However, there is potential for considerable savings in tne testing and maintenance phases. Once sufficient data is collected, COCOMO may have to be recalibrated to reflect this. This may cause changes in the constants of the COCOMO equations. The distribution of effort throughout the software life cycle will change as well.

Barry Boehm reports [Boe86] that TRW has studied the impact of Ada upon software development costs. Three studies show that, at the beginning of 1985, use of Ada increased software development costs by factors ranging from 1.2 to 2.5. The three studies were consistent in predicting that Ada software development costs will decrease rapidly as Ada experience accumulates, reaching the break-even point (a factor of 1.0) between mid-1987 and mid-1989, and leveling off at a factor of about 0.75 (that is, a 25% cost saving) around the end of 1991. (These figures only address development costs. Use of Ada is also expected to drastically reduce maintenance costs.)

The studies are based on the assumption that the following COCOMO cost drivers will decrease as experience with Ada increases:

- Language experience

- o Support-environment experience
- o Support-environment stability
- o Support environment completeness and integration
- o Use of modern programming practices
- o Execution time
- o Storage efficiency

Considering that over two-thirds of the 63 data points used to validate COCOMO were either FORTRAN or assembly-language efforts, the COCOMO equations may have to be recalibrated for Ada developments. TRW has been studying this problem, but has not released its findings publicly. With experience, the COCOMO model may also have to be adjusted to consider the impact of reusable software, which was apparently not a factor in the three TRW studies.

12.2 Software Life Cycle Model (SLIM)

The Software LIfecycle Model (SLIM) is a proprietary model developed by L.H. Putnam and currently offered by Quantitative Software Management, Inc. The Jensen model described in Section 12.3 is based on Putnam's work and bears strong resemblance to SLIM. Putnam [Put80] describes SLIM as a comprehensive package that not only estimates software costs, but also provides a vehicle for effective planning and management of software development.

SLIM estimates the effort beginning with the detailed design of a system's programs and ending with the completion of acceptance testing. It is assumed that the system requirements and the system specifications are completed in activities that precede the development cycle, although a front-end option is available that calculates time and effort for the feasibility study and functional design activities. Both direct and indirect hours are included in the development effort.

SLIM is based primarily on a function describing levels of effort over the software life cycle. The function can be applied to both the total life cycle effort and individual component activities, such as design, coding, and validation. SLIM uses powerful mathematical analysis tools, including the PERT sizing technique, linear programming, Monte Carlo simulation, and risk analysis, to build upon the level-of-effort function and estimate software costs.

SLIM has three primary input parameters that strongly influence the effort estimate:

- o System size is described in terms of the number of executable source language statements to be written. The size estimate is calculated one way during the early phases of tne life cycle and another way when the system functions have been defined. In either case a probability distribution is derived for a range of possible size values.

- o Level of difficulty is measured by the amount of interfacing, new design, and concurrent programming that will go on during development.

- o A technology constant describes the system development environment. The constant represents several system characteristics, including the language used, the tools used, target machine, use of modern programming practices, development-computer availability, human skills, complexity of the system, and customer interfaces. It is through this constant that the model is calibrated to a particular environment.

Once the effort estimate is derived from these input parameters, the development cost in dollars in computed.

SLIM is currently available on-line through the time-sharing facilities of American Management Systems' AMShare service. Further information about this service can be obtained from:

American Management Systems
1777 North Kent Street
Arlington, VA 22209

(800) 336-4786

The proprietary nature of SLIM makes it difficult to assess how or if SLIM should be recalibrated for Ada.

12.3 Jensen's Software Development Resource Estimation Model

Randall W. Jensen [Jen??] of Hughes Aircraft Company has developed a software development resource estimation model that extends earlier macro-level estimation work done by Putnam [Put80], Norden, and others. The Jensen macro-estimation model applies to projects of five or more people lasting no more than five years. It provides estimates for costs beginning after the requirements definition phase is complete and extending through test and integration.

The model is driven by four primary cost parameters:

- o Effective size specified as lines of source code

- o The complexity of the problem

- The technical ability of the developer to produce the product (the technology constant)

- The ability to staff the project (staffing rate)

As with COCOMO, the estimation of the number of lines of source code is a principle input, but there is no guidance on how to derive this number for Ada software development efforts. It is also not clear what constitutes a line of source code. (Does each executable statement count as a line, regardless of how the text is formatted? What about declarations and pragmas? What about blank lines, comments, and lines containing a single keyword?) Jensen does provide rating tables for the other three cost drivers.

12.3.1 Jensen's Computations

Jensen's model is based on three equations:

- An equation relating project staffing after a certain amount of elapsed time to the total project effort in man-years and the total project development time in years

- An equation relating complexity to total project effort and total project development time.

- The software equation, relating effective system size in source lines to the effective developer technology constant, the total project effort, and the total project development time

Using the second and third equations, Jensen derives a fourth equation, giving total project effort in terms of complexity, number of source lines, and the technology constant.

The equations are applied as follows:

1. The technology constant is computed, as will be described below.

2. The fourth equation is applied to compute total project effort. Jensen assumes that 40% of this effort is attributable to software development.

3. Given the total project effort, the second equation is applied to predict total development time.

4. The first equation is applied to predict project staffing over time.

The effective developer technology constant is closely related to the adjustment factors developed by Boehm for COCOMO. The parameters include:

- Analyst capability
- Application experience
- Programmer capability
- Modern development practices
- Automated development practice support
- Turnaround time from editing of code through retrieval of a hard copy

Ratings are assigned to each of these categories as in COCOMO. In cases where the categories are the same, Jensen uses the same values as Boehm. Jensen uses thirteen environmental factors to augment the technology-constant considerations:

- Special display requirements
- Detail and stability of operational requirements
- Real-time constraints
- CPU-time constraints
- Memory constraints
- Virtual-machine experience
- Virtual-machine volatility
- Use of a remote computer by the developer
- Whether development takes place at the operational site
- Whether development takes place on a computer other than the target computer
- Whether development takes place at multiple sites
- Programming-language experience
- System reliability

There is no readily available literature on the performance of the Jensen model. In the absence of corroborative data, there is reason to doubt the model's predictions: First, many of the inputs influencing the complexity factor are subjective. Second, the relationships described by Jensen's equations may be oversimplistic. Third, the attribution of 40% of the total project effort to software development appears arbitrary.

12.3.2 Use of the Jensen Model to Estimate Ada Software Development Efforts

The impact of Ada on Jensen's estimation model is similar to its impact on COCOMO. Both use the same rating numbers for such cost drivers as programmer capability, language experience, use of modern programming practices, use of software tools, space and time constraints, support-environment experience, and support-environment stability. As explained in Section 12.1.4, these cost factors are likely to decrease over the next few years as experience with Ada increases.

The Jensen model, like COCOMO, needs to have several factors recalibrated for Ada development efforts. For example, Ada's support for software reuse may reduce the impact of special display requirements once packages have been developed for screen manipulation, windows, and graphics. The proposed Common APSE Interface Set (CAIS) standard [CAIS85] takes a step in this direction by defining packages for three classes of terminals with varying capabilities, called scroll, page, and form terminals. (See Section 3.4 for a further discussion of the CAIS.)

In addition, the computation of staff levels over time should be closely compared to actual Ada development efforts to determine its validity. Because Ada software development may involve higher levels of effort in early phases of the project, Jensen's first equation may no longer apply.

12.4 PRICE-S

The PRICE-S model is the software version of RCA's parametric cost modeling series. Like the other PRICE models, PRICE-S is a commercially available model that can be accessed through On-Line Systems, Inc., a time-sharing network. The principal factors influencing the PRICE-S output are the number of source instructions, the class of application, the complexity of the problem, and the resources available for development.

RCA believes that the use of Ada will affect several of the cost driving parameters. Schedules, at least in the near term, will most likely be longer, for the following reasons:

- Some Ada implementations have no tool support beyond the bare minimum needed to compile, link and execute an Ada program.
- Some projects will be required to write their own libraries and general utilities, thus extending the schedule.

- Ada is still not widely taught at the university level, so the requirement of learning a new language is thrust into the project schedule.

The PRICE-S complexity variable is being adjusted to account for these factors. As more data becomes available, RCA will continue to make adjustments to provide better forecasting.

RCA expects that the longer schedules will be accompanied by more expensive labor rates. People knowledgeable in Ada are commanding a higher price in the marketplace today, because of their small numbers. The PRICE-S resources variable has been adjusted to account for this condition. RCA also believes that use of Ada will alter the distribution of labor throughout the life cycle. However, there is not yet enough empirical data about this to draw specific conclusions.

Further information about PRICE-S can be obtained from:

Mr. Earl King
RCA PRICE Systems
300 Route 38, Building 146
Moorestown, NJ 08057

(609) 866-6567

12.5 Workshop on COCOMO Cost Estimates for WIS

Section 12.1 described the fifteen COCOMO cost drivers. A January 1985 Government/industry workshop held at the Institutes for Defense Analyses [DSP85] focused on Ada's impact upon these cost drivers of COCOMO and upon WIS (WWMCCS (WorldWide Military Command and Control System) Information System) maintenance costs. The following questions were addressed:

1. How is Ada different from traditional programming languages?

2. Should the fifteen cost drivers be defined for and applied to Ada as they have been for other languages?

3. Which cost drivers are language-independent?

4. Should Ada or WIS multipliers be factored into the equations?

5. What is Ada's impact on lines of code, the primary cost driver of COCOMO?

6. What is Ada's impact on traditional scheduling techniques?

Since most of the previously developed WIS software is written in COBOL, most of the discussions compared Ada to COBOL. The conclusions were largely speculative, and consensus could not

always be reached. The body of the report contains no empirical data to support the participants' opinions.

The conclusions were as follows:

- Using Ada instead of COBOL would result in a 44% reduction of effort for application programs and 40% reduction of effort for support software.
- Initially, there will be an increase in costs due to the lack of familiarity with Ada and immature Ada implementations.
- Of the fifteen cost multipliers, only two, database size and complexity and application experience, will show no appreciable difference using Ada.
- Five cost multipliers will show reductions in both the near term and the far term:
  - Required system reliability
  - Software product complexity
  - Turnaround time on the development computer
  - Use of modern programming practices
  - Use of software-development tools
- The virtual-machine experience multiplier will increase significantly in both near-term and far-term efforts.
- The analyst-capability and programmer-capability multipliers will decrease significantly in the far term, but with little change in the near term.
- A 75% reduction in maintenance costs will result from using Ada instead of COBOL.

Workshop participants also identified factors that call into question the applicability of COCOMO to WIS:

- The WIS management structure is very complex, staffed by all of the services, with many chains of command. The many and varied WIS customers often disagree about requirements. COCOMO assumes well-defined requirements and well-managed projects with little slack time. The scale of management may increase costs anywhere from 15% to 300%.
- The WIS program is expected to have volatile requirements, primarily due to new capabilities spawning other requirements by the end users. Until requirements are specified in an unambiguous manner, they actually specify a spectrum of

products that could be developed, each with its own complexity and cost. COCOMO assumes stable requirements.

- COCOMO does not consider the large number of programmers who need to be trained in Ada. This has significant cost implications, and is beyond the scope of software estimation models.

SECTION 13

BENCHMARKS

13.1 Environment Issues

13.1.1 Evaluation

Evaluation of an Ada Programming Support Environment (APSE) consists of assessing characteristics such as usability, efficiency, and maintainability of an entire APSE or an individual APSE component (such as a compiler or an editor). The requirements for evaluation can be divided it two major groups: component requirements and macroscopic requirements for interactions among the components. The APSE evaluation should also include evaluation of the human interface and the data interfaces among tools. A well-defined set of component characteristics definitions, as well as general information on the evaluation process, can be found in an APSE evaluation and validation requirements document [E&V84] prepared by the Evaluation and Validation Team Requirements Working Group for the Ada Joint Program Office.

When assessing an APSE, an organization should investigate the degree to which the APSE supports the organization's software development metnodology. It is also important to evaluate the flexibility and extensibility of the environment to support multiple methodologies.

13.2 Ada Compiler Evaluation Capability (ACEC)

13.2.1 Purpose and Scope

Under contract to the Ada Joint Program Office APSE Evaluation and Validation (E&V) team, the Institute for Defense Analyses produced the Prototype Ada Compiler Evaluation Capability (ACEC) [RHV86]. The ACEC consists of an organized suite of compiler performance tests, as well as support software for executing these tests and collecting performance analysis data.

The Prototype ACEC provides two ways of evaluating an Ada compiler. A user may select a set of tests that measures the frequency distribution of language constructs in a real application. Alternatively, the user could execute all tests in order to gain some insight regarding the language features which could pose a stress load for the given compiler/run-time combination, were these features among the most frequently used. These measurements are not absolute performance metrics of the efficiency of a particular compiler architecture. They are only an indication of the effect produced by an Ada feature when used in a particular compiler/run-time environment combination.

13.2.2 Test Categories and Attributes

The Prototype ACEC test units fall into two major categories: normative and optional. The normative tests provide information about the standard language features, while the optional category provides information about combinations of language constructs and/or compiler features that might be of interest for a particular application.

The normative tests determine the system cost for a particular language feature. There are two normative test subcategories: performance (a group of tests that collect speed and space attributes), and capacity (a group of tests that indicate the limitations imposed by the compiler and the run-time system on the application).

The optional tests represent the features most frequently used by the application at hand. This category is also subdivided in two subcategories: features (testing non-standard Ada language features and certain compiling options), and special algorithms (testing combinations of language constructs characteristic of certain applications, such as the Whetstone synthetic benchmark programs and the Sieve of Eratosthenes).

The user is provided with a descriptive information about each test through the test's attributes:

- A description of the test objective
- The category of the test, including major category and subcategory
- The evaluation criteria for the test
- The language feature being tested
- Whether this is a test, a control program, or the test with an optimization feature
- The kind of statistics being collected

More information about the Prototype ACEC can be obtained from:

DoD IDA Management Office
1801 North Beauregard Street
Alexandria, VA 22311

The prototype ACEC is a good first step in compiler evaluation. In many cases the testing of the application-independent features should be balanced with testing of application-specific features. These tests are supplied by the user and give more relevant measurements of time and memory constraints.

SECTION 14

SOFTWARE LIBRARIES

As noted in Section 11.2, libraries of Ada reusable software components are a prerequisite to software reuse. These libraries can be developed and maintained internally, obtained from a vendor, or acquired from the public domain. Section 14.1 discusses two commercial Ada software libraries. Section 14.2 discusses the Ada Repository, a public-domain library.

14.1 Commercial Ada Libraries

This section describes two commercially available libraries of reusable Ada software. The first, GRACE, consists primarily of general-purpose data-structure manipulation packages. The second, Math Advantage, is a traditional scientific-subroutine library like those that have long been used by FORTRAN programmers.

14.1.1 GRACE

EVB Software Engineering is producing a series of general-purpose software parts called GRACE (an acronym for Generic Reusable Ada Components for Engineering). These parts can only be acquired by subscribing to the entire series, to which new parts are added periodically. Currently the series contains 133 parts. By the end of 1986, EVB Software Engineering is planning to double this number.

The scope of the GRACE parts is open-ended, but the early emphasis has been on elementary data structures. Currently available GRACE parts deal with simple data structures, such as stacks, queues, lists, and matrices, and the most common algorithms associated with these structures. GRACE parts are classified according to a taxonomy developed by Grady Booch [Boo86] for reusable Ada software. This taxonomy focuses on the software's time and space requirements rather than its functional behavior.

Further information can be obtained from:

EVB Software Engineering, Inc.
451 Hungerford Drive, Suite 701
Rockville, MD 20850

(301) 251-1626

14.1.2 Math Advantage

Quantitative Technology Corporation has produced an Ada version of its Math Advantage scientific subroutine library, including operations for floating-point, integer, and complex vectors and matrices, signal-processing operations, and image-processing operations. The library was previously written in FORTRAN and C, and the Ada version was released in June 1986. The Ada version appears to be a direct translation of the Ada and C versions. Unfortunately, this means that the library does not exploit features unique to Ada and does not follow accepted Ada style conventions. All subprogram names are abbreviations of six characters or less, no use is made of operator overloading (for example, a routine named CRVADD rather than the + operator is used to add two complex vectors), and the underlying numeric types appear to be restricted to the predefined types INTEGER and FLOAT, restricting the precision of the operations to some implementation-dependent value.

Further information can be obtained from:

Quantitative Technology Corporation
8700 SW Creekside Place, Suite D
Beaverton, OR 97005

(503) 626-3081

14.2 The Ada Repository

The Ada Repository is a collection of Ada programs, tools, educational material, and other information maintained by the Department of Defense since November 1984 on the SIMTEL-20 host of the Defense Data Network. Files can be obtained either over the network or on tape. Reusable parts currently in the Ada repository include:

- General purpose data-structure manipulation packages, including string-handling packages
- Packages of mathematical routines
- Database-manipulation packages
- Ada source code for text editors

- Form-generation tools
- File manipulation tools for paged files

Besides these reusable parts, the Ada repository also contains Ada development tools, including program formatters, cross-reference tools, and program complexity-measurement tools; Ada training materials; and general information about Ada.

Many contributions to the Ada Repository come from Defense Department organizations, notably the Air Force WIS program and the Naval Oceans Systems Center. Some of these contributions were developed in-house and some under contract. Universities, corporations, and individuals have also volunteered contributions to the Repository.

Further information on the Ada Repository can be obtained from:

Commander, U.S. Army White Sands Missile Range
STEWS-CD-MS
(Bldg. 362, Frank wancho)
White Sands, NM 88002-5072

Those with access to the Defense Data Network can be placed on an electronic mailing list by sending a request to ADA-SW-REQUEST@SIMTEL20. A master index to the Ada Repository is available in looseleaf form from:

Echelon, Inc.
885 North San Antonio Road
Los Altos, CA 94022

(415) 948-3820

LIST OF REFERENCES

[A&M85] Anderson, Christine M., and McNicholl, Daniel G. Reusable software -- a mission-critical case study. AIAA/ACM/NASA/IEEE Computers in Aerospace V Conference, Long Beach, California, October 1985, 136-139

[Boe81] Boehm, Barry W. Software Engineering Economics. Prentice-Hall, Englewood-Cliffs, New Jersey, 1981

[Boe86] Boehm, Barry W. Welcoming remarks, SIGAda meeting, Los Angeles, California, February 1986

[Boo82] Booch, Grady. Object-oriented design. Ada Letters **1**, No. 3 (March-April 1982), 64-76

[Boo86] Booch, Grady. Software Components with Ada. Benjamin-Cummings, Menlo Park, California, in production

[CAIS85] Military Standard Common APSE Interface Set (CAIS). Proposed MIL-STD-CAIS, January 31, 1985

[Coh85] Cohen, Norman H. The SofTech Ada verification project. AIAA/ACM/NASA/IEEE Computers in Aerospace V Conference, Long Beach, California, October 1985, 399-407

[Coh86] Cohen, Norman H. MAVEN: the Modular Ada Validation ENvironment. In Mayfield, W. Terry, ed., Preliminary Proceedings of the Third Ada Verification Workshop, Research Triangle, North Carolina, May 1986, 2-1 to 2-18

[DSP85] Douville, Anne A., Salasin, John, and Probert, Thomas H., eds. The Impact of Ada on COCOMO Cost Estimates as Applied to the Worldwide Miltary Command & Control (WWMMCCS) Information System (WIS). Institute for Defense Analyses, 1985

[Dye83] Dyer, M. Software validation in the cleanroom method. Technical report TR 86.0003, IBM Federal Systems Division, Bethesda, Maryland, August 19, 1983

[E&V84] Requirements for Evaluation and Validation of Ada Programming Support Enivironments, Version 1.0. Evaluation and Validation Team Requiremetns Working Group, October 17, 1984

[Kri86] Krieg-Bruckner, B., et al. Program development by specification and transformation in Ada/Anna. In Ada: Managing the Transition. Proceedings of the Ada-Europe International Conference, Edinburgh, 6-8 May 1986. Cambridge University Press, Cambridge, U.K., 1986, 249-258

[Lie86] Lieblein, Edward. The Department of Defense software initiative -- a status report. Communications of the ACM **29**, No. 8 (August 1986), 734-743

[MRY86] McDonald, Catherine W., Riddle, William, and Youngblut, Christine. STARS methodology area summary, volume II, software life cycle and methodology selection. ACM Software Engineering Notes **11**, No. 2 (April 1986), 58-65

[N&W84] Nissen, John, and Wallis, Peter, eds. Portability and Style in Ada. Cambridge University Press, Cambridge, U.K., 1984

[Par72] Parnas, D.L. On the criteria to be used in decomposing systems into modules. Communications of the ACM **15**, No. 12 (December 1972), 1053-1058

[Put80] Putnam, Lawrence H. Software Cost Estimating and Life Cycle Control: Getting the Software Numbers. IEEE Computer Society Press, New York, 1980

[RHV86] Riccardi, Gregory A., Hook, Audrey A., and Vilot, Michael J. Ada compiler performance benchmark. In Peter J.L. Wallis, ed., Ada: Managing the Transition, Proceedings of the Ada-Europe International Conference, Edinburgh, 6-8 May 1986, Cambridge University Press, Cambridge, U.K., 1986, 33-41

[Sof84] Ada portability guidelines. Hq, Electronic Systems Division/XRSE, Hanscom AFB, MA 01731. ESD-TR-85-141, ADA160390.

BIBLIOGRAPHY

Ada portability guidelines. Report 3285-2-208/1, SofTech, Inc., Waltham, Massachusetts

Anderson, Christine M., and McNicholl, Daniel G. Reusable software -- a mission-critical case study. AIAA/ACM/NASA/IEEE Computers in Aerospace V Conference, Long Beach, California, October 1985, 136-139

Baker, Paul L. Experience with the integration of Ada design methods. Proceedings of the 4th Annual National Conference on Ada Technology, Atlanta, Georgia, March 1986, 51-56

Boehm, Barry W. Software Engineering Economics. Prentice-Hall, Englewood-Cliffs, New Jersey, 1981

Boehm, Barry W. Welcoming remarks, SIGAda meeting, Los Angeles, California, February 1986

Booch, Grady. Object-oriented design. Ada Letters **1**, No. 3 (March-April 1982), 64-76

Booch, Grady. Software Components with Ada. Benjamin-Cummings, Menlo Park, California, in production

Burton, Bruce, and Broido, Michael. Development of an Ada package library. Proceedings of the 4th Annual National Conference on Ada Technology, Atlanta, Georgia, March 1986, 42-50

Carrio, Miguel A., Jr. The technology life cycle and Ada. Proceedings of the 4th Annual National Conference on Ada Tecnnology, Atlanta, Georgia, March 1986, 75-82

Cohen, Norman H. The SofTech Ada verification project. AIAA/ACM/NASA/IEEE Computers in Aerospace V Conference, Long Beach, California, October 1985, 399-407

Cohen, Norman H. MAVEN: the Modular Ada Validation ENvironment. In Mayfield, W. Terry, ed., Preliminary Proceedings of the Third Ada Verification Workshop, Research Triangle, North Carolina, May 1986, 2-1 to 2-18

Douville, Anne A., Salasin, John, and Probert, Thomas H., eds. The Impact of Ada on COCOMO Cost Estimates as Applied to the Worldwide Miltary Command & Control (WWMMCCS) Information System (WIS). Institute for Defense Analyses, 1985

Dyer, M. Software validation in the cleanroom method. Technical report TR 86.0003, IBM Federal Systems Division, Bethesda, Maryland, August 19, 1983

Evaluation of Ada programming environments. Joint Missiles Project Office, Washington, D.C., May 1984

Firesmith, Donald G., and Gilyeat, Colin B. Resolution of Ada-related concerns in DoD-STD-2167, Revision A. ACM Ada Letters **4**, No. 5 (September-October 1986), 29-33

Freeman, Peter, and Wasserman, Anthony I. Software methodologies and Ada. Ada Joint Program Office, November 1982

Holdsworth, D. Prototyping, production and portability. Ada User **7**, No. 1 (1986), 58-62

Jensen, Randall W. The art of software development schedule and cost estimation, Hughes Aircraft Company, no date

Krieg-Bruckner, B., et al. Program development by specification and transformation in Ada/Anna. In Ada: Managing the Transition. Proceedings of the Ada-Europe International Conference, Edinburgh, 6-8 May 1986. Cambridge University Press, Cambridge, U.K., 1986, 249-258

Lieblein, Edward. The Department of Defense software initiative -- a status report. Communications of the ACM **29**, No. 8 (August 1986), 734-743

McDonald, Catherine W., Riddle, William, and Youngblut, Christine. STARS methodology area summary, volume II, software life cycle and methodology selection. ACM Software Engineering Notes **11**, No. 2 (April 1986), 58-85

Military Handbook Defense System Software Development Handbook. DoD-HDBK-287 (draft), MCCR 0010, May 1986

Military Standard Common APSE Interface Set (CAIS). Proposed MIL-STD-CAIS, January 31, 1985

Military Standard Defense System Software Development. DoD-STD-2167, MCCR N3608, June 1985

Military Standard Software Quality Evaluation. DoD-STD-2168 (draft), April 1985

Nissen, John, and Wallis, Peter, eds. Portability and Style in Ada. Cambridge University Press, Cambridge, U.K., 1984

Parnas, D.L. On the criteria to be used in decomposing systems into modules. Communications of the ACM **15**, No. 12 (December 1972), 1053-1058

Putnam, Lawrence H. Software Cost Estimating and Life Cycle Control: Getting the Software Numbers. IEEE Computer Society Press, New York, 1980

Requirements for Evaluation and Validation of Ada Programming Support Enivironments, Version 1.0. Ada Joint Program Office Evaluation and Validation Team Requiremetns Working Group, October 17, 1984

Requirements for Evaluation and Validation of Ada Programming Support Environments, Version 2.0. Ada Joint Program Office Evaluation and Validation Team Requirements working Group, October 1986

Riccardi, Gregory A., Hook, Audrey A., and Vilot, Michael J. Ada compiler performance benchmark. In Peter J.L. Wallis, ed., Ada: Managing the Transition, Proceedings of the Ada-Europe International Conference, Edinburgh, 6-8 May 1986, Cambridge University Press, Cambridge, U.K., 1986, 33-41

Roman, Gruia-Catalin. A taxonomy of current issues in requirements engineering. IEEE Computer **18**, No. 4 (April 1985), 14-23

Roski, Steve. "DoD-STD-2167 default design and coding standards. ACM Ada Letters **4**, No. 5 (September-October 1986), 34-44

Ross, Douglas T. Applications and extensions of SADT. IEEE Computer **18**, No. 4 (April 1985), 25-34

User's Manual for the Prototype Ada Compiler Evaluation Capability (ACEC) Version 1. Paper P-1879, Institute for Defense Analyses, October 1985

APPENDIX

Points of Contact for Ada Information

Ada Repository	Commander U.S. Army White Sands Missile Range STEWS-CD-MS (Bldg. 362, Frank Wancho) White Sands, NM 88002-5072 Echelon, Inc. 885 North San Antonio Road Los Altos, CA 94022 (415) 948-3820
AdaGRAPH	The Analytic Sciences Corporation 1700 North Moore Street Suite 1220 Arlington, VA 22209 (703) 558-7400
AMShare (SLIM cost-estimating model on-line)	American Management Systems 1777 North Kent Street Arlington, VA 22209 (800) 336-4786
Benchmarks	(see Prototype ACEC)
Byron and DARTS II	Intermetrics, Inc. 733 Concord Avenue Cambridge, MA 02138 (617) 661-1840

DoD-STD-2167 (package of Air Force guidelines)	Roland Usher U.S. Air Force Electronic Systems Division ESD/PLEA Hanscom AFB, MA (617) 377-4002
GRACE	EVB Software Engineering, Inc. 451 Hungerford Drive, Suite 701 Rockville, MD 20850 (301) 251-1626
Math Advantage subroutine library	Quantitative Technology Corporation 8700 SW Creekside Place, Suite D Beaverton, OR 97005 (503) 626-3081
PAMELA	George W. Cherry P.O. Box 2429 Reston, VA 22090 (703) 427-4450
PRICE-S	Mr. Earl King RCA PRICE Systems 300 Route 38, Building 146 Moorestown, NJ 08057 (609) 866-6567
Prototype ACEC	DoD IDA Management Office 1801 North Beauregard Street Alexandria, VA 22311
SHARP	Lt. Mark V. Ziemba ESD/PLSE Hanscom AFB, MA AUTOVON 478-2656 MILNET Ziemba.SDruid@ RADC-MULTICS.ARPA
SLIM	(see AMShare)